FlashTrie: Beyond 100-Gb/s IP Route Lookup Using Hash-Based Prefix-Compressed Trie

Masanori Bando, *Associate Member, IEEE*, Yi-Li Lin, and H. Jonathan Chao, *Fellow, IEEE*

***Abstract*—It is becoming apparent that the next-generation IP route lookup architecture needs to achieve speeds of 100 Gb/s and beyond while supporting IPv4 and IPv6 with fast real-time updates to accommodate ever-growing routing tables. Some of the proposed multibit-trie-based schemes, such as TreeBitmap, have been used in today's high-end routers. However, their large data structures often require multiple external memory accesses for each route lookup. A pipelining technique is widely used to achieve high-speed lookup with the cost of using many external memory chips. Pipelining also often leads to poor memory load-balancing. In this paper, we propose a new IP route lookup architecture called *FlashTrie* that overcomes the shortcomings of the multibit-trie-based approaches. We use a hash-based membership query to limit off-chip memory accesses per lookup and to balance memory utilization among the memory modules. By compacting the data structure size, the lookup depth of each level can be increased. We also develop a new data structure called *Prefix-Compressed Trie* that reduces the size of a bitmap by more than 80%. Our simulation and implementation results show that FlashTrie can achieve 80-Gb/s worst-case throughput while simultaneously supporting 2 M prefixes for IPv4 and 318 k prefixes for IPv6 with one lookup engine and two Double-Data-Rate (DDR3) SDRAM chips. When implementing five lookup engines on a state-of-the-art field programmable gate array (FPGA) chip and using 10 DDR3 memory chips, we expect FlashTrie to achieve 1-Gpps (packet per second) throughput, equivalent to 400 Gb/s for IPv4 and 600 Gb/s for IPv6. FlashTrie also supports incremental real-time updates.**

***Index Terms*—DRAM, field programmable gate array (FPGA), FlashTrie, hash, IPv4, IPv6, longest prefix match, membership query, next-generation network, PC-Trie, Prefix Compressed Trie, route lookup.**

I. Introduction

RECENTLY, major Internet carriers and vendors succeeded in experimenting with 100-Gb/s equipment. In September 2008, Verizon successfully performed 100-Gb/s transmission for more than 646 mi. Comcast and Cisco also announced successful trials of 100-Gb/s router interfaces in June 2008. Juniper announced a 100-Gb/s line card for their core router (T1600) in March 2010 [2]. AT&T and Cisco announced successful completion of the world's first field trial of 100-Gb/s backbone network technology, which took place in AT&T's live network between New Orleans, LA, and Miami, FL, using Cisco's latest carrier router, CRS-3, in March 2010 [3]. Accordingly, the IEEE 802.3ba Working Group is expected to complete the standard for 100-Gb/s Ethernet by 2010 [4]. For a 100-Gb/s link, 250 million lookups per second are required in the worst case. In other words, we have only 4 ns to complete one lookup, and exceeding the 100-Gb/s lookup speed remains a challenge.

We can also observe continuous growth of the Internet by observing the total amount of traffic. Cisco's global IP traffic forecast report states that global IP traffic will double nearly every two years by the end of 2012, and thus, annual global IP traffic will exceed half a zetta (10^{21}) bytes by 2013 [5]. The report indicates that not only are throughput requirements of the core networks increasing, but routing tables are also expanding exponentially [6] and are expected to grow to 1–2 million routes in the near future. In addition to these requirements, next-generation routers must also support IPv4 and IPv6 seamlessly along with real-time update capabilities. This trend challenges current routers, especially core routers, by demanding support for higher throughput and larger routing tables.

In this paper, we propose *FlashTrie*, a low-cost, high-speed scalable route lookup architecture that eliminates the shortcomings of currently available schemes by using: 1) on-chip membership query, and 2) a new data structure called *Prefix-Compressed Trie*. FlashTrie can achieve route lookup for a 400-Gb/s link for IPv4 with 2 M routes and an IPv6 with 318 k routes by using a single field programmable gate array (FPGA) and 10 DRAM chips. This small number of required devices results in low cost and low power consumption. FlashTrie also supports incremental real-time updates. This paper also discusses two optimizations and prototype hardware implementations.

Section II discusses previously proposed architectures and identifies their pros and cons. Section III gives an overview of the scheme and introduces two methods that can resolve the current issues. Section IV describes the detailed architecture and query process. Section V discusses issues and our solutions for updating. Section VI demonstrates the performance and optimization of the FlashTrie, and improvements are shown in Section VII. An overview and the results of a prototype hardware implementation are presented in Section VIII. Section IX concludes the paper.

II. Related Work and Motivation

In IP route lookup, the system extracts each incoming packet's destination IP address and performs the longest prefix match. Ternary content-addressable memory (TCAM)-based schemes [7], [8] are widely used in midrange routers. However, their high cost and large power consumption make them

Manuscript received April 25, 2010; revised April 16, 2011 and September 30, 2011; accepted November 06, 2011; approved by IEEE/ACM Transactions on Networking Editor T. Wolf. Date of publication March 14, 2012; date of current version August 14, 2012. This work is an extended version of papers presented at the IEEE International Conference on Computer Communications (INFOCOM), San Deigo, CA, March 15–19, 2010.

M. Bando and H. J. Chao are with the Department of Electrical and Computer Engineering, Polytechnic Institute of New York University, Brooklyn, NY 11201 USA (e-mail: mbando01@students.poly.edu; chao@poly.edu).

Y.-L. Lin is with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan (e-mail: yllin@csie.ncku.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2012.2188643

1063-6692/$31.00 © 2012 IEEE

unattractive for core routers. Direct lookup schemes [9]–[11] can use standard SRAM or DRAM to store the next hop for each prefix in a table or multiple tables that are addressed by the prefix. This scheme is only effective for short address lookups (e.g., less than 16 bits), but will not be realistic for longer lookups due to *prefix expansion* [12].

To avoid the prohibitively large memory requirements, hash-based schemes [13]–[20] have been proposed. Whether applying a hash function to each prefix length or to a certain prefix length (e.g., /16, /24 and /32 for IPv4), those prefixes are hashed to a table. Various methods have been proposed to reduce the number of prefixes hashed to the same entry of the hash table. Bloom filters are sometimes used to query the existence of the prefix before finding the next-hop information (NHI) of the prefix [16], [18].

Hardware trie-based schemes [1], [21]–[26] can achieve high throughput. However, they require many memory chips in parallel to accommodate the pipeline stages required by the many levels of the trie, which has a height proportional to the number of bits in the IP address. This is especially a problem for IPv6, which has larger number of bits in the address. Multibit-trie architectures [1], [22]–[25], such as Tree Bitmap [1], have gained much attention because they can reduce the number of pipeline stages and because of their efficient data structures. Each Tree Bitmap node contains two pieces of information: the Internal Bitmap of the subtrie and a pointer for the NHI; and the External Bitmap for a head pointer to the block of child nodes and a bitmap for child subtries. As a result, one lookup requires multiple off-chip memory accesses. To reduce the number of off-chip memory accesses, Song *et al.* proposed Shape Shift Tries (SST) [23], which allows the number of trie levels in each access to be flexible. SST can achieve approximately 50% reduction in memory accesses compared to the Tree Bitmap. Although this reduction is significant, the number of memory accesses required by the SST is still considerable. In addition, SST is only suitable for sparse tries, limiting its application to future routers.

A different way to reduce memory accesses in the Tree Bitmap architecture is to increase the stride size. However, this will increase the bitmap size exponentially and result in more off-chip memory accesses that limit system performance. Another disadvantage for choosing a large stride size is that update speed may be degraded. This is because there will be more child nodes in each trie, and they are stored in consecutive memory locations. Whenever a new child node is added, many other child nodes are moved to other memory locations. In the worst case, the entire block of child nodes is relocated.

Another typical drawback of trie-based schemes is their uneven distribution of data structures in memory. Usually in the tries, the lower level contains many more prefixes than the higher level. Each pipeline stage consists of either one level or multiple levels in the trie and typically stores the information of its prefixes in a memory bank. As the number of prefixes differs drastically from stage to stage, the loading among memory modules is quite uneven, resulting in low memory utilization. A scheme called CAMP proposed by Kumer *et al.* states a solution for this memory uneven problem [27], where several subtries share one pipeline stage (memory). Although this scheme can achieve a higher memory utilization, the lookup performance degrades from possible memory access contention. In [25], the authors proposed a solution to balance the pipeline memory. However, their scheme requires 25 independent memory chips resulting in a prohibitively high cost. The number of memory chips required is even more when IPv6 is supported.

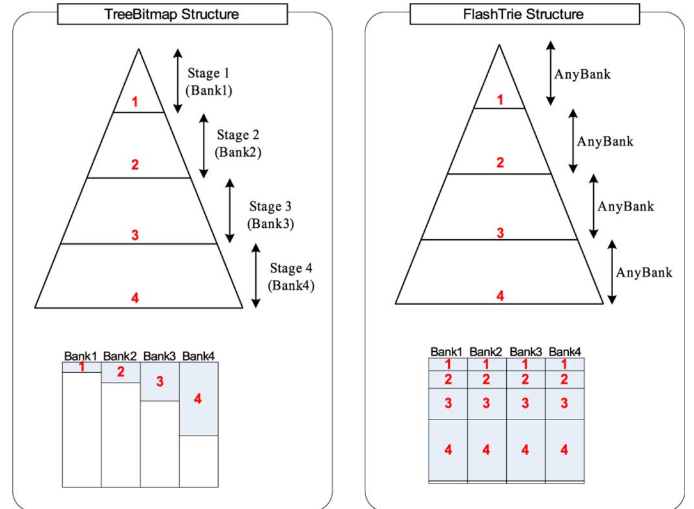

Fig. 1. DRAM bank allocation of TreeBitmap and FlashTrie.

For instance, as shown in Fig. 1, a DRAM chip has four memory banks with each storing tree bitmap information of each level. Each arriving packet goes through the four pipeline stages (i.e., the four levels of multibit tries) to complete the lookup, ending up traversing the banks one after the other before the next packet can start the lookup operation. As a result, only a packet can access the memory within a memory cycle. On the contrary, with FlashTrie, each packet will only need to access one of the four levels of multibit tries, resulting in only accessing to one of the four memory banks. FlashTrie also organizes memory well so that all levels of the multibit tries can fit in one DRAM bank. Therefore, up to four packets can access to the memory within a memory cycle. In conclusion, FlashTrie achieves four times the throughput of the TreeBit-based lookup schemes.

III. FlashTrie

A. Overview

An overview of the FlashTrie architecture with an example routing table is shown in Fig. 2. The routing table has 10 routes for which corresponding prefixes and NHI are shown in the table. The binary trie for this routing table is constructed next to the routing table. We divide the binary trie into different *levels* based on these k-bit subtries ($k = 4$ in this example). Thus, Level 0 contains from prefix length *0* to prefix length *3*, Level 1 contains from prefix length *4* to prefix length *7*, and so on. Each level contains one or more subtries.

It is important to note that all subtries should be independent among different levels in the FlashTrie architecture to guarantee only one off-chip memory access of the multibit subtrie. Once the prefix has been found in the multibit subtrie, an additional off-chip memory access of NHI is followed, which can be pipelined, and thus the lookup speed of one packet per memory access is achieved. For example, in the figure, the subtrie that contains prefixes P7 and P9 does not have NHI in the root node. This means, empty nodes (nodes that are not related to P7 and P9) in this subtrie depend on the node present one level up (P4

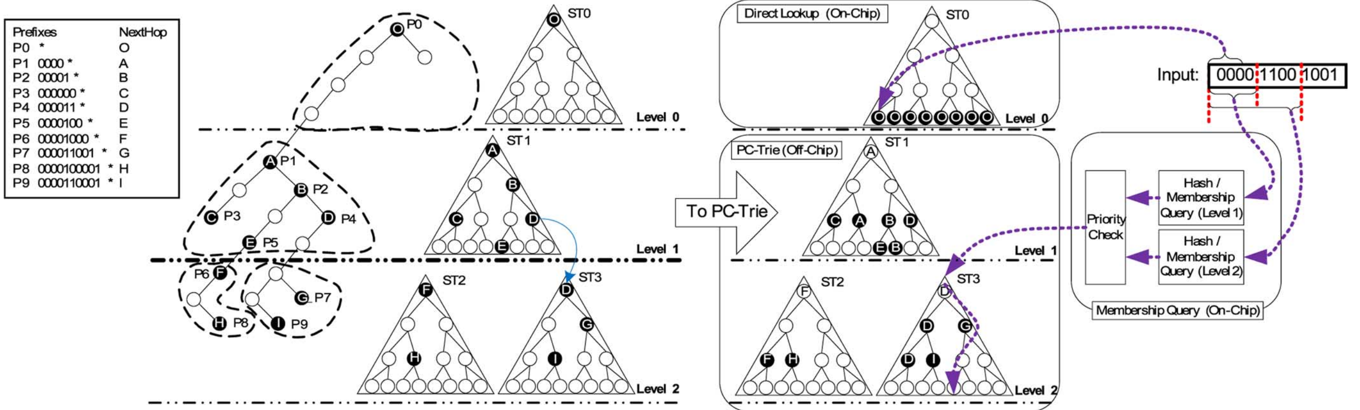

Fig. 2. Overview of FlashTrie architecture.

in the example). To remove this dependency, we copy the NHI of P4 to the root of the subtrie (ST3) illustrated as an arrow.

The subtries are converted to a new compressed trie called *Prefix-Compressed Trie* (PC-Trie), described in Section III-B, which is then stored in off-chip memory. In the actual system, all root nodes are stored in on-chip memory, which facilitates easy updating (as described in Section V). The top level (Level 0) uses a direct lookup approach, so ST0 is not a PC-Trie and is stored in on-chip memory.

One of the most important advantages of the FlashTrie architecture is that it guarantees only one off-chip memory access to resolve IPv4/IPv6 trie. To ensure this, lightweight on-chip hash modules are deployed to perform membership queries. An optimized hash data structure, called HashTune [19], is used for the hash function. The hash data structure is discussed in Section III-C-II. The hash tables are queried to find the existence of a subtrie at each level in parallel. Since there are multiple levels, there could be matches in more than one level at a time. This is resolved based on priority (the longest prefix has the highest priority). Thus, only the longest matching subtrie is obtained from off-chip memory. The number of off-chip memory accesses is controlled by this on-chip membership operation. This operation is illustrated on the right side of the figure. Section IV explains our new compressed data structure PC-Trie and *membership query operation*.

B. Prefix-Compressed Trie

Controlling the number of memory accesses per lookup (ideally, one memory access) is realized by managing bitmap size. Reducing the bitmap size is one of the main motivations for this work. Current DRAM has, at most, a 16-bit data bus and a burst size of 8, so one DRAM access can read or write, at most, 128 bits [28], [29]. Thus, any bitmap size exceeding 128 bits requires multiple memory accesses that significantly degrade lookup performance (speed). In Tree Bitmap, the internal bitmap has $2^{\mathrm{stride}} - 1$ bits and the external bitmap consumes 2^{stride} bits. Thus, the 9-bit stride size requires more than 1 kb, which requires multiple off-chip memory accesses. The Tree Bitmap scheme does not involve any bitmap compression technique. Hence, the bitmap size increases exponentially. Although the Tree Bitmap scheme proposes two optimizations, split tree bitmap and segmented bitmap, they are not sufficient. Using split tree bitmap, the internal and external bitmaps are stored in separate memories. This way, the Tree Bitmap node is reduced to nearly half the actual size. Still, one bitmap size is too big to be fetched from an off-chip memory access. With segmented bitmap, the original bitmap is cut in half each time it is segmented. However, each segmented node must have a pointer, which eventually creates considerable overhead. As a result, segmented bitmap optimization actually increases the total memory requirement. These two optimizations, as well as other Tree Bitmap types of data structures, suffer from the same problem.

We propose a new data structure named PC-Trie, as illustrated in Fig. 3. The main difference between Tree Bitmap and our data structure is that a bit in the Tree Bitmap represents only one node, while a bit represents more than one node in PC-Trie. More specifically, one bit of the PC-Trie can represent consecutive nodes (called siblings) in the same prefix length.

The example shows one subtrie that includes five prefixes $(*, 1*, 00*, 11*, 100*)$ and the corresponding NHI $(\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E})$. In *Step 1*, the routing table is simply translated into a binary trie representation. Fig. 3(a) shows Tree Bitmap for the given routing table. Since Tree Bitmap simply converts the binary trie representation to the bitmap representation, *Steps 2* and *3* are the same as in the binary trie. Bit positions in the bitmap are set to "1" at the locations that have a prefix, and set to "0" otherwise, as shown in the *Final Data Structure* in the figure. Now let us look at PC-Trie2 in Fig. 3(b). It illustrates the conversion process from a binary trie to PC-Trie2. The suffix (the number) represents the compression degree. PC-Trie2 means compression of two sibling nodes into one. The two sibling bits that are compressed are marked by a dotted circle. Let us denote this set of nodes as *node sets*.

Construction of the PC-Trie has two rules: 1) all sibling nodes must be filled with NHI if at least one node in the node set contains NHI; 2) the parents node set can be deleted if two child node sets exist. Let us start applying rule 1. Rule 1 says that if the sibling is not present, NHIs are copied and filled with either the parent's or the ancestor's NHI. Look at the PC-Trie in *Step 2*. Prefixes C, D, and E are the only children with an empty sibling. The empty siblings of C, D, and E need to be filled with their respective parent's or ancestor's NHI. In the example, A is the parent of empty sibling C, and B is the parent of empty sibling D and the grandparent of empty sibling E. Thus, *Step 3* shows all the empty siblings filled with their respective parent's or ancestor's NHI. Applying rule 2, a node set that contains A

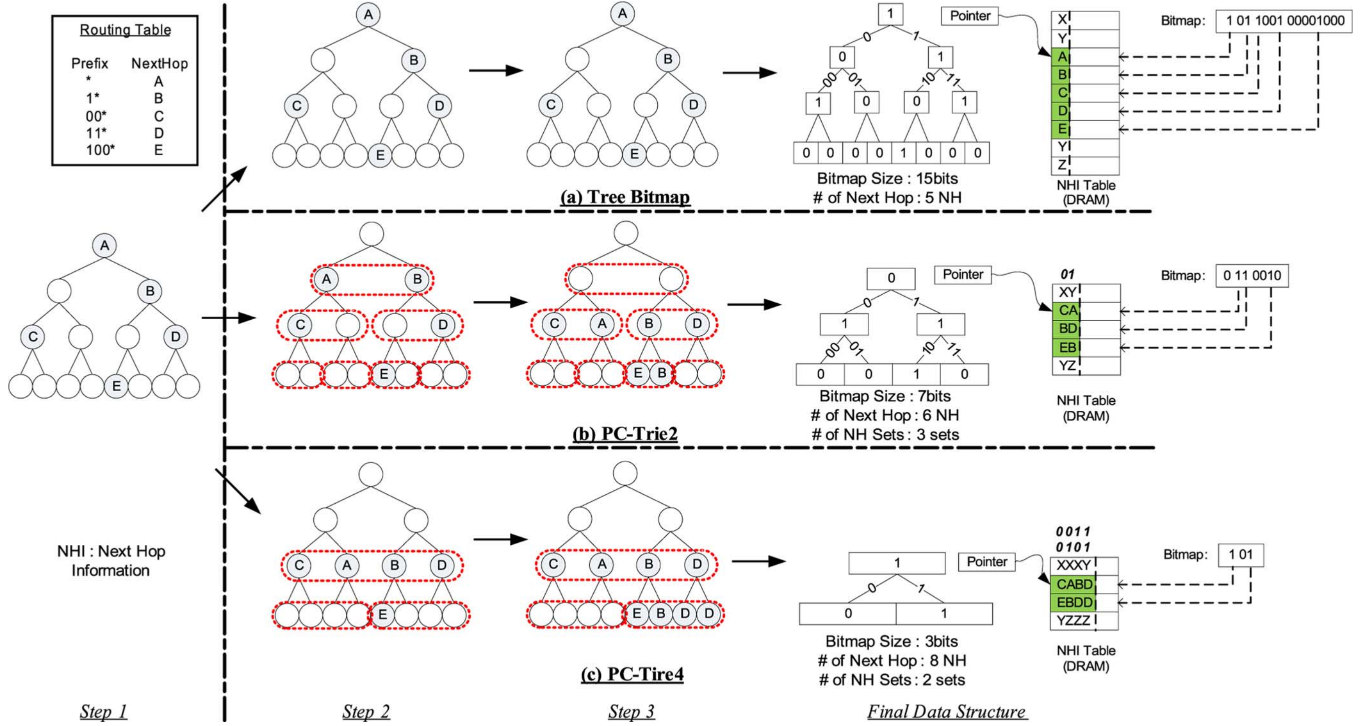

Fig. 3. Bitmap transformation from binary trie to (a) Tree Bitmap, (b) Prefix-Compressed Trie 2, and (c) Prefix-Compressed Trie 4.

and B can be eliminated. In the *Final Data Structure* step, the bitmap is constructed from the PC-Trie. Only one bit is required to represent two nodes. As a result, the bitmap size is reduced from 15 to 7 bits (a reduction of more than half). Similarly, a higher degree of compression can be achieved to further reduce the bitmap size. The bitmap size for the n degree of compression can be formulated as PC-Trie $n = 2^{(s-\log_2(n))} - 1$ bits where s is the stride size (in bits). Thus, for a PC-Trie 8 and 9-bit stride $(n = 8, s = 9)$, the PC-Trie requires *63 bits* as compared to *511 bits* for the Tree Bitmap. Construction procedures of PC-Trie4 are also illustrated in Fig. 3(c).

One drawback of compressing the bitmap is that NHI may need to be duplicated as the figure shows. For instance, six NHIs are needed for the compressed PC-Trie2, while the original number of NHIs is five. However, the number of memory slots needed for the NHI table is reduced. For example, as shown in Fig. 3, Tree Bitmap needs five slots, while PC-Trie2 needs only three. The size of an NHI is so small that multiple NHIs can be stored into a single memory slot. A single DRAM read (e.g., 128 bits) for a memory slot effectively fetches multiple NHIs. This not only results in a more compact memory footprint, but more importantly, the number of memory accesses required is the same as that before compression, if not fewer. Both IPv4 and IPv6 NHI can simultaneously be fit in one bank of a DRAM chip after compressing the bitmap, as shown in Section VI. A high-compression rate is also available in PC-Trie. Algorithm 1 shows the pseudocode to construct PC-Trie as explained above.

Let us consider the query process of input "100" as an example. According to the routing table shown in the figure, the PC-trie must return "E" as the next-hop information of the corresponding input. The same result can also be obtained by traversing the binary trie. The Tree Bitmap data structure is shown in the right side of Fig. 3(a). First, check the bitmap

Algorithm 1: Prefix-Compressed Trie Construction

1: **Prefix-Compressed Trie (SubTrie[ ], stride, compSize)**
2: //All SubTries
3: **for** ($i = 0$ to SubTrie.Length; i++) **do**
4: //All PC-Trie node
5: **for** ($s =$ compSize-1 to 2^{Stride}; $s = s +$ compSize) **do**
6: //All nodes in a PC-Trie node
7: **for** ($t = s$ to $s +$ compSize-1: t++) **do**
8: **if** (At least one prefix exist in an CompNode) **then**
9: [Fill the PC-Trie node with proper NHI]
10: **end if**
11: **end for**
12: **end for**
13: //Eliminate Redundancy
14: **for** ($ns = 0$ to $ns =$ compSize; ns++) **do**
15: **if** (A PC-Trie node has both child PC-Trie nodes.) **then**
16: [Remove the PC-Trie node.]
17: **end if**
18: **end for**
19: **end for**

contents of "100," where "1" represents the existence of the next-hop information. By using a pointer and the offset value (number of 1's prior to the "100" location), the corresponding NHI "E" can be retrieved from the NHI memory. Now, let us explain the query procedure of the PC-Trie as shown in Fig. 3(b) and (c). For simplicity, we use PC-Trie2 to explain the concept. The query processes of PC-Trie4 and higher perform in a similar manner. The query input "100" is divided into two

parts. The first part is used for bitmap access, and the second part is used to identify the corresponding NHI. For PC-Trie2, the input except for the least significant bit is used for bitmap access (first part, "10" in this example). The least significant bit (second part, "0" in this example) is used for NHI selection. When traversing the PC-Trie with the input of "10," the result of "1" is reached, indicating the existence of NHI. A pointer and the offset value are then used to access a set of NHI, "EB," from the NHI memory. Finally, the second part of the bit "0" is used to select the correct NHI, "E." In other words, PC-Trie returns multiple NHIs from the NHI memory, and a corrected NHI is chosen based on the least significant bit(s). The last step of NHI selection is an extra step from the Tree Bitmap scheme, but gives us an advantage of having a larger lookup depth in levels (i.e., a larger stride size).

C. Membership Queries

As mentioned earlier, one of the most significant features of FlashTrie is only one off-chip memory access for IPv4/IPv6 trie. To ensure this, each off-chip memory access must return the intended PC-Trie for the queried input IP address. Otherwise, additional memory accesses are required to determine the longest matching prefix in the upper level of the subtrie. FlashTrie performs on-chip membership queries to achieve this. The most popular architecture to perform membership queries is the Bloom filter [30], which is referenced in several IP lookup papers [16], [18]. The most recognized advantage of the Bloom filter is that the result is free from false negatives. However, the result still contains false positives. False positives can be reduced by increasing the number of hash functions per lookup, the size of the hash table, or both. Achieving a lower false-positive rate requires a considerable amount of resources and many parallel hash functions that increase system complexity and downgrade system performance. Even after all the effort, the false-positive rate will still not converge at zero. We overcome this issue by using an exact match operation along with a hash function. Each entry of the hash table holds all or a portion of the root IP address of the programmed subtrie. We call this entry *verify bits* and perform an exact matching operation with the input IP address. Hash functions inherently do not have any false negatives. By means of an exact matching operation, we ensure no false positives as well.

1) Membership Query Module: The basic function of the membership query module is to take an IP address as input, process it, and return a corresponding PC-Trie address. A block diagram of the membership query module is shown in Fig. 4. In the programming phase, all subtries are hashed, and the contents of hash tables are constructed in off-line. We use HashTune [19] as the hash function because it has a compact data structure and better memory utilization, which is explained in Section III-C.2. Since we use a hash function, there are possible collisions. Therefore, the hash table has two types of entries: one each for collision and noncollision cases as shown in the figure. If the hash table entry has a collision, then its Least Significant Bit (LSB) is set to "1"; else, it is set to "0" for no collision. The collided items are stored in Black Sheep (BS) memory located in the membership query module.

For querying operation, the input IP address is hashed, and whether this hash entry has a collision or not is determined by checking the LSB of the hash table entry. In the noncollision

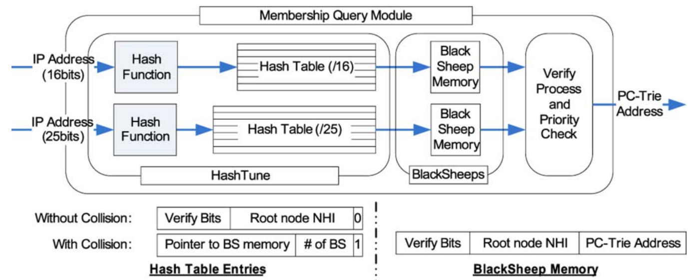

Fig. 4. Block diagram of membership query module.

case, the hash table entry contains *Verify Bits* and *Root node NHI*. If the Verify Bits are matched with the input IP address, then the hash result becomes the PC-Trie address for the input IP address. Thus, the PC-Trie addresses are not stored in the hash tables. In the collision case, the hash table entry has a pointer to BS memory and the number of collided items in the bin. Since a hash may result in one or more collisions, we store the number of collisions for each hash table entry. In the case of more than one collision, the BS memory is required to be accessed multiple times if only one BS memory module is used. This can become the bottleneck of the system. Instead, we have multiple on-chip BS memory modules that are accessed in parallel. Based on our simulation results, in the worst case, three BS memory modules are needed for IPv4 and 14 BS memory modules are needed for IPv6. Fig. 4 also shows BS memory entry contents that are *Verify Bits*, *Root Node NHI*, and *PC-Trie Address*. Thus, whichever Verify Bits of the BS memory entry are matched with the input IP address, the corresponding PC-Trie address is retrieved from the BS memory entry. With the on-chip memory-based collision resolving architecture, we can avoid using area- and power-consuming TCAMs or content-addressable memories (CAMs). By this membership query operation, only one PC-Trie address is resolved.

Hash-based matching schemes may encounter a small possibility of hash collisions, where the number of elements stored in the same bin exceeds the bin size. A straightforward solution is to minimize the probability by using a small on-chip CAM to store the overflow elements [31]–[33]. However, on-chip CAM requires a significant amount of resources and power. On the other hand, using on-chip RAM suffers from nondeterministic query time due to a link-list addressing approach. Our proposed Black Sheep Memory scheme uses collision avoidance and parallel operations to eliminate this drawback.

2) HashTune: In FlashTrie, memory-efficient HashTune [19] is used for hashing operations. In contrast with a naive hash function, HashTune has two important advantages: 1) key distribution is more uniform over the entire hash table; and 2) the size of Verify Bits can be reduced. Mitzenmacher shows the advantage of using more than one hash function [34]. The main contribution of his paper is an approach to balance queue length by selecting the shortest queues from multiple one. HashTune allows individual groups to select different hash functions if keys in the group do not distribute well (for example, if the number of collisions exceeds a predefined threshold, use another hash function and rehash the keys). Hao *et al.* proposed a hash function that reduces collisions using the grouping idea [35]. Each group is allowed to select a different hash function to maintain

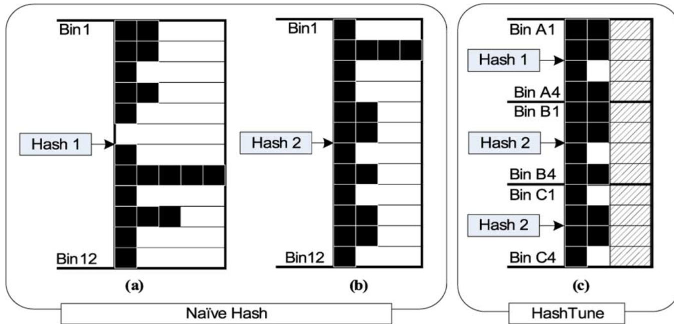

Fig. 5. HashTune example. (a) Key distribution using a naive hash function (Hash 1). (b) Key distribution using a naive hash function (Hash 2). (c) Key distribution using HashTune.

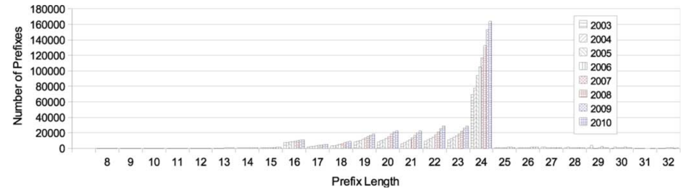

Fig. 6. IPv4 prefix distribution from 2003 to 2010.

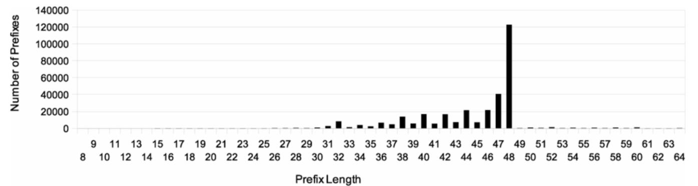

Fig. 7. IPv6 prefix distribution.

minimum collisions. The scheme uses a hash function to distribute keys among the multiple groups while our scheme uses a few bits of the keys. Contrasted to the proposed hash function, our scheme has an advantage in terms of memory requirements. Since all keys in a group share the same group ID, the redundant group ID bit can be omitted from verify bits, which saves a large amount of memory.

Fig. 5(a) and (b) shows key distribution using a naive hash approach. Each row represents a bin, and each dark square in the row shows a key hashed to that bin. Typically, a naive hash function leads to the nonuniform distribution of items in the hash table. This nonuniformity forces unnecessarily large bin sizes. Even when a good hash function is found for a particular table, after some updates, the distribution of items in the hash table can still become nonuniform. Fig. 5(c) shows key distribution using HashTune.

In HashTune, the entire hash table is segmented into multiple small hash tables called *groups*. All groups have the same number of bins. In Fig. 5(c), there are 12 bins segmented into three groups, and each group has four bins (GroupA: Bins A1–A4, GroupB: Bins B1–B4, GroupC: Bins C1–C4). Each group is allowed to select a different hash function from the pool of hash functions. GroupA uses Hash 1, and GroupB and GroupC use Hash 2 in the example. The selected hash function ID is stored in a Hash ID table, which is also stored in on-chip memory and used for query operations. After applying different hash functions, occupancy of bins in each group and, hence, in the entire hash table becomes even. Because of the balanced key distribution, the hash table size is smaller than in naive approaches, indicated by shaded areas in the figure.

After the prefix updates, if the hash function assigned to a bin group no longer provides good uniform distribution of items, HashTune replaces that hash function with a more suitable function from the hash pool, i.e., rehashing. Note that this rehashing only affects a small portion of the items, those that are hashed to a particular bin group and can be handled in a reasonable time without disturbing the performance [19].

The second advantage is also derived from grouping. Each group is assigned an ID called *Group ID*, and the ID is selected from several bits of the root node in each subtrie. For example, an 8-bit group ID will be selected for 256 groups. We select the group ID from the LSBs of the root node because this balances the number of keys per group [19]. The bits assigned for the group ID can be taken out from the Verify Bits because all items in the group have the same the group ID. For example, resolving 17 bits input and choosing 8 LSBs as the group ID gives us the remaining 9 bits to be stored as Verify Bits. As a result, the Verify Bit size and on-chip memory requirements are reduced.

IV. FlashTrie Architecture and Lookup

In this section, we analyze the actual routing table to construct the FlashTrie architecture. We first study prefix distributions of routing tables and determine the coverage and size (stride size) of subtries. Fig. 6 shows the prefix distribution for IPv4 based on the length of the prefix for the period 2003–2010 using the routing table obtained from the RouteViews project [36], Oregon. All routing tables are snapshots of the same day (January 29) in each year. This history of prefix distribution shows the trend is not changing.

One of the characteristics of the distribution is the number of prefixes in /24, which is more than 50% of the total number of prefixes. Any multibit-trie-based architecture attempts to maintain fewer subtries so that memory consumption is less. Assigning /24 in a root of the subtrie is not a good idea because it requires the number of subtries to be equal to the number of prefixes in /24. Thus, we place /24 at the bottom of the subtrie. We select three levels based on the prefix distribution of Fig. 6. They are IPv4/15 [15 Most Significant Bits (MSB) of an IPv4 address], IPv4/16 (MSB 16–24 bits), and IPv4/25 (MSB 25–32 bits). IPv6 prefix distribution is also considered using real routing tables [36] and expected future IPv6 routing tables [37]. Prefix distribution of the expected future IPv6 routing tables is shown in Fig. 7. We can observe that majority of prefixes is in /48. We place /48 at the bottom of the subtrie. Next, we describe the overall FlashTrie architecture and explain the IP route lookup process.

A. Architecture and Lookup Operation

Fig. 8 illustrates the flow from receiving incoming packets until obtaining the NHI for the input IP address using IPv4 route lookup architecture. The IPv6 lookup procedure is similar except that there are more levels.

The input 32-bit IPv4 address is categorized in IPv4/15, IPv4/16, and IPv4/25. IPv4/15 is resolved using Direct Lookup (on-chip), and IPv4/16 and IPv4/25 are resolved using the membership query module (as explained in Section III-C.1) (on-chip) and PC-Trie (off-chip). We resolve the PC-Trie

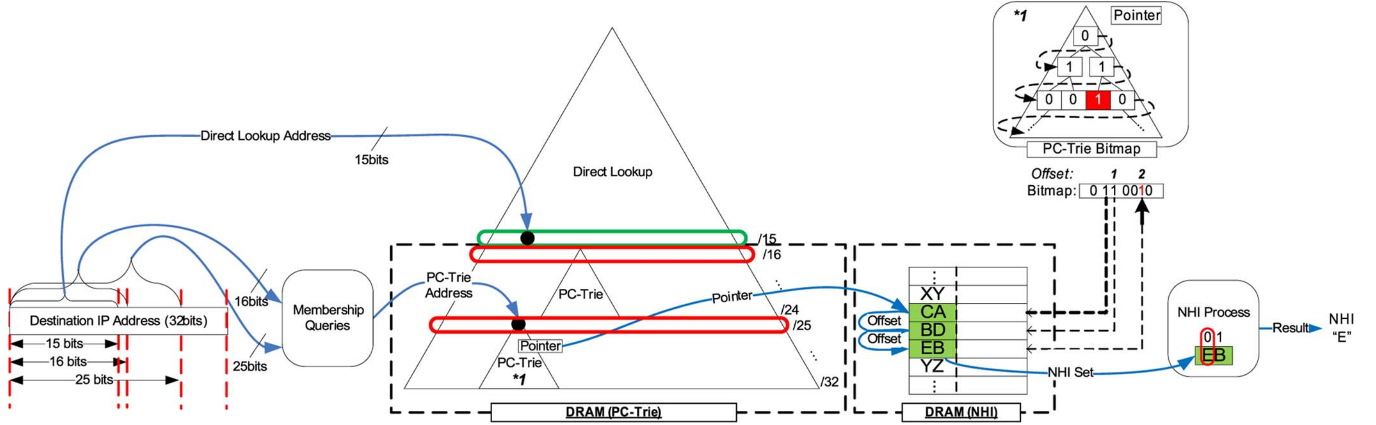

Fig. 8. FlashTrie architecture for IPv4.

address, marked *1 in the figure, from the output of the membership query module. Suppose "100" is the input to this PC-Trie. We can simply traverse the PC-Trie bitmap as we do in a binary trie. If the bit is "0," then we go to the left child; else we go to the right child. We start traversing from the MSB of input. The aim is to find the longest matching prefix in the PC-Trie. After traversing "1" (Right) and "0" (Left), we end up with the third bit in the bottom (dark square in the PC-Trie). The content of the bitmap is "1," which means NHI exists for the input "100." Since this is the longest matching prefix, we resolve the address of NHI memory for this node set. The address is resolved by a pointer stored with the PC-Trie node and offset calculated from the bitmap. The offset is the number of 1's starting from the root of the PC-Trie until the longest matching location. In this case, the offset is 2 (as there are two 1's before the final location). The pointer is pointing to the beginning of NHI entries in the memory for this PC-Trie. In the example, the pointer points to the memory location of CA. The offset is added to this memory location to get the exact NHI set EB. Finally, the NHI is selected from the NHI set by the LSB of the input. Since the LSB of the input is " 0," **E** is selected as the final NHI. If LSB is "1," **B** is selected. For PC-Trie$n$ or a higher Prefix-Compressed Trie, one NHI set contains four or more NHIs. In this case, more than one $(\log_2(n))$ bit from the input destination IP address is used to find the final NHI.

V. Update

Online update capability is an important criterion for future routers. Currently, there are four updates observed per second on average, but 25 k updates per second during peak periods [6]. For the update analysis, we take peak rate to demonstrate worst-case update capability. The anticipated growth in routing table entries is 2 million, which is approximately 6.3 times the current amount of routing table entries. (This anticipation is made based on the routing table growth reported by the RouteViews project [36] as well as the CIDR report [6].) If updates also increase proportionately, there would be around 160 k $(=25\ \text{k} * 6.3)$ updates per second during peak periods. To support this kind of growth, we reserve 10 million packets per second (Mpps) for update operations. This is sufficient for both on-chip and off-chip memory updates, as the Section VI explains.

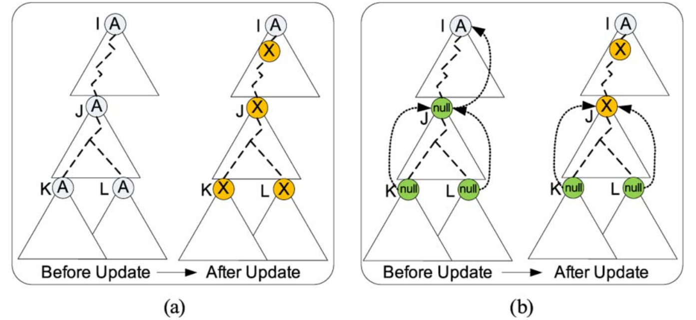

Fig. 9. On-chip memory update. (a) Direct NHI programming. (b) Indirect NHI programming.

1) On-Chip Memory Update: NHI of the root at each subtrie is stored in on-chip memory as described in Section III. Fig. 9 shows two types of update methods: 1) direct NHI programming, and 2) indirect NHI programming. The example in Fig. 9 shows four root nodes of subtries represented as I, J, K, and L. For *direct NHI programming*, since there is no intermediate node between each subtrie root, NHI Ⓐ of root node I is simply copied to the root nodes, J, K, L of the following level. A problem occurs when an update is required at some upper stream along the path. For instance, if a new route Ⓧ is to be added in the subtrie that has a root node I on the path, then all the following root nodes (J, K, L) also need to be updated. The number of updates might be excessively high, especially for IPv6 because of longer addresses (more levels). To avoid this problem, we propose *Indirect NHI programming* as shown in example (b), in which *null* value is stored at the root node that does not have NHI to indicate actual NHI located in the root node of upper level. While doing so, the update of Ⓧ traverses only one level down. Thus, the new route Ⓧ only affects the root node J, but not K, L. It is important to note that by making sure there is only one level of dependency, we maintain all the intermediate subtries in the on-chip membership query module. In other words, if some intermediate subtries do not exist, we create only root node NHI and store it in the membership query module (i.e., if a subtrie exists at Level(l), root NHIs of the upper level from Level$(l-1)$ to Level(1) exist in the membership query module).

An example of an on-chip update using the indirect NHI programming method is illustrated in Fig. 10. The figure illustrates four levels of subtries, the root NHI supposed to be stored in on-chip memory. In the example, update is applied in the subtrie

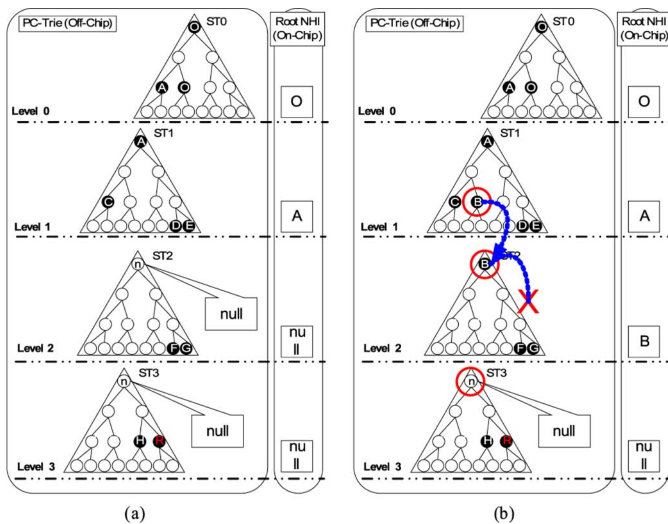

Fig. 10. On-chip memory update detail. (a) Before update applied. (b) After an update is applied in Level 1.

in Level 1, and a new route B is added. The update to subtries is one level down (Level 2), so the root node of ST2 is modified from *null* to B. Any update only affects one level below, and the far level does not have any influence. Indeed, the root node of ST3 located in Level 3 maintains the same NHI, *null*.

In the worst case for an 9-bit stride, there can be 511 updates (if the node below the root node has to be updated). Therefore, for future 160 k updates with each update requiring 511 memory accesses, the total updates, in the worst case, will be 81.8 million $(=511\times160\ \text{k})$, which is less likely to be happen. Since this is an on-chip hash table update, 10 Mpps is still sufficient because the on-chip hash table is segmented and stored in individual on-chip memories. Multiple hash tables can be updated independently.

2) Off-Chip Memory Update: There are two independent off-chip memories—one for Prefix-Compressed Trie (PC-Trie) and another for NHI. For PC-Trie, we need eight memory accesses per update because the same PC-Trie is duplicated among all 8 banks. This requires 1.28 Mpps (= 8 banks × 160 k updates). For NHI, the worst-case number of memory accesses needed per update is $(\text{Stride} - \log_2(\text{Degree of Compression})) \times (\text{number of bins})$ (e.g., for PC-Trie8 with 9-bit stride, six memory accesses are required). If PC-Trie8 is used for all subtries, then 7.68 Mpps are needed (6 memory access × 8 banks × 160 k updates) in the worst case. Thus, 10 Mpps is more than enough to update both on-chip and off-chip memory in real time.

In addition, Tree Bitmap update is not trivial, especially when inserting new child nodes. This operation requires multiple memory accesses to arrange all child nodes in consecutive memory locations. Moreover, after multiple updates, Tree Bitmap has to perform defragmentation to make consecutive memory space available. On the other hand, FlashTrie does not have these problems because it systematically organizes hash-based addressing. Items stored in Black Sheep memory hold individual pointers, and therefore, we can allocate PC-Trie without any constraint.

VI. Performance Evaluation

For evaluation purposes, we obtain current routing tables from multiple locations such as Oregon; California; London, U.K.; and Tokyo, Japan. These routing tables have similar prefix distributions. Therefore, we use the largest routing table, Oregon, from the RouteViews project [36] (Jan 01 2010 00:00). This table contains 318 043 routes. We generate 2 million routes for the future routing table based on the prefix distribution trend discussed in Section IV. The size of a real IPv6 routing table is still around 3000 [6], [36], which is extremely small for the evaluation. We synthesize an anticipated IPv6 routing table following the methods presented in [37]. It uses IPv4 routes and an autonomous system (AS) number, giving a more realistic routing table as compared to a randomly generated table. Thus, the synthesized IPv6 routing table contains 318 043 routes (same as the real IPv4 routing table size).

TABLE I
Routing—Number of Subtries in Each Level for the Three Routing Tables

Level	IPv4 (Real)	IPv4 (2M)	Level	IPv6
/16 (16-24bits)	22,545	51,968	/13 (13-21bits)	33
/25 (25-32bits)	4,504	9,373	/22 (22-30bits)	233
			/31 (31-39bits)	6,768
			/40 (40-48bits)	117,298
			/49 (49-57bits)	7,372
			/58 (58-63bits)	2,562
Total Sub-tries	27,049	61,341		134,266

First, we extract subtries to generate the FlashTrie data structure. Table I shows the number of subtries in each level and the total subtries for all three routing tables. We mainly use an 9-bit stride size for the entire evaluation, and detail stride settings are listed in Table I. This level selection is based on the observations of real routing table characteristics discussed in Section III. We also show the distribution of IPv4 routing table from 2003–2010 in Fig. 6, from which we conclude that it is safe to assume that this trend will not change severely. Considering the long-term stability of the IPv4 distribution trend, we believe that the IPv6 distribution trend will also be steady. Even if this trend changes, the deployment transition is slow and gradual. Our DRAM-based approach has enough capacity to absorb such changes. Following this setting, we evaluate performance in terms of memory requirements and lookup speed for the IPv4 and IPv6 routing tables. The evaluation is based on the results of simulation and hardware implementation.

A. Memory Requirements

In FlashTrie, two types of memory are used: 1) on-chip memory, and 2) off-chip memory. We next discuss the data that is stored in memory and the memory size required to support IPv4 and IPv6 simultaneously.

1) On-Chip Memory: FlashTrie pushes most of the memory-consuming operation outside of the chip (off-chip memory). Some operations, however, are kept on-chip to enhance lookup performance and online updateability. On-chip memory is used for: 1) direct lookup for up to /15(IPv4) or /12(IPv6); 2) hash table for membership queries; 3) hash ID table (storing a hash ID for each group); and 4) Black Sheep memories for collided items in the hash table. The on-chip memory requirements are shown in Table II. The first 15 bits are resolved by the direct lookup approach in IPv4 with 8 bits NHI ($256\ \text{kb} = 2^{15}$ bits). We use 64 hash functions for all tables and levels. Taking IPv4/16 from

TABLE II
ON-CHIP MEMORY REQUIREMENTS

	IPv4 (Real)	IPv4 (2M)	IPv6
# of Prefixes	318,043	2M	318,043
Direct Lookup	256Kbits	256Kbits	32Kbits
Hash Table	0.32Mbits	1.11Mbits	2.1Mbits
Hash ID Table	15Kbits	60Kbits	43Kbits
Black Sheep Memory	0.53Mbits	0.05Mbits	6Mbits
Total	1.11Mbits	1.47Mbits	8.17Mbits

TABLE III
CONFIGURATIONS FOR NUMBER OF GROUPS AND NUMBER OF BINS IN EACH GROUP. ($Gr.$: $Group$)

	IPv4 (Real)		IPv4 (2M)			IPv6	
	#ofGr.	Gr.Size	#ofGr.	Gr.Size		#ofGr.	Gr.Size
/16	2,048	8	8,192	8	/13	32	8
/25	512	8	2,048	8	/22	128	8
					/31	512	8
					/40	4,096	8
					/49	2,048	8
					/58	512	8

TABLE IV
PREFIX-COMPRESSED TRIE BITMAP MEMORY REQUIREMENT COMPARED WITH TREE BITMAP

	IPv4(Real)	IPv4(2M)	IPv6
# of Sub-Trie	18,537	60,125	118,867
Tree Bitmap	18.8Mbits	61.0Mbits	120.5Mbits
PC-Trie 2	4.5Mbits	14.8Mbits	30.8Mbits
PC-Trie 4	2.4Mbits	7.9Mbits	16.5Mbits
PC-Trie 8	1.4Mbits	4.5Mbits	9.3Mbits
PC-Trie 16	0.9Mbits	2.8Mbits	5.8Mbits

the real routing table as an example, the hash table is segmented into 2048 groups. Therefore, the hash ID table size will be 12 kb ($= 2048 \times \log_2(64)$ bits). The simulation result determines the number of groups and bins in each group. The configurations are summarized in Table III.

It is clear from the results in Table II that to support 2 M IPv4 routes (1.47 Mb) and 318 k IPv6 routes (8.17 Mb) simultaneously, we need 9.64 Mb of on-chip memory. Thus, we can have three copies of this data in the on-chip memory of a state-of-the-art FPGA that has 38 Mb on-chip memory capacity. Also, all on-chip memory has independent dual ports. Therefore, six engines can fit on one chip.

2) Off-Chip Memory: The off-chip memory (DRAM) in FlashTrie consists of two independent memory chips, PC-Trie and NHI memory. Table IV shows memory requirements for PC-Trie. The subtrie size for Tree Bitmap is 1063 bits (internal bitmap + external bitmap + two pointers), and for FlashTrie it is 83 bits (PC-Trie bitmap + pointer) for 9-bits strides PC-Trie8. This significant reduction is because of bitmap compression and the elimination of the external bitmap. Table V shows memory requirements for the NHI (assuming 8 bits NHI). The PC-Trie for every level can fit in one bank of DRAM as shown in Fig. 11(a). Even if NHI is duplicated, we can still easily fit IPv4 and IPv6 NHI into one bank of DRAM as shown in Fig. 11(b). The data allocation in the figure is based on a 1-Gb memory (each bank has 128 Mb with 8 banks). The graph in Fig. 12 shows the off-chip memory required for bitmap and NHI for different degrees of compression of PC-Trie in the FlashTrie as compared to the Tree Bitmap. It is evident from the result that the reduction in bitmap size is

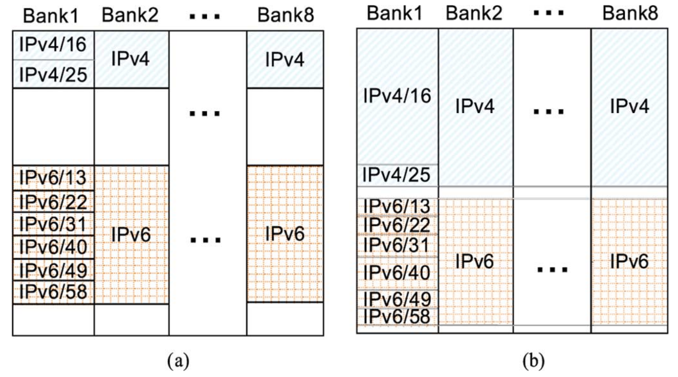

Fig. 11. DRAM memory data allocation. (a) PC-Trie memory. (DRAM) (b) NHI memory (DRAM).

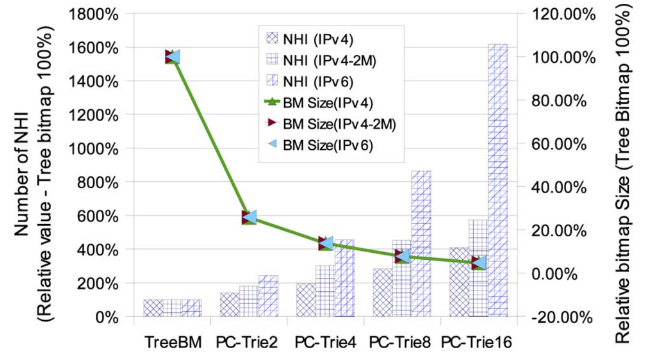

Fig. 12. Off-chip memory requirements of TreeBitmap versus FlashTrie.

more than 80% (for higher compression degree of PC-Trie). As another comparison, let us show a result of smaller stride size in TreeBitmap scheme. For a given smaller stride size, the number of subtries increases because the coverage of each subtrie reduces. As a result, the bitmap memory requirements also increase. Using a real 318 k IPv4 routing table, there are 75 041 subtries with a stride-5 setting. (11 490 subtries in IPv4/15; 60 801 subtries in IPv4/20; 2438 subtries in IPv4/25; and 312 subtries in IPv4/30. The length represents root node level.). A stride of 5 is a reasonable setting for the TreeBitmap scheme since one bitmap can fit (103 bits) in one DRAM burst access (128 bits). In total, 12.1 Mb are required in an off-chip memory (9.6 Mb for the bitmap memory and 2.5 Mb for the NHI memory.) With a similar off-chip memory size, FlashTrie can achieve a much higher throughput as shown in this section.

B. Lookup Speed and Timing Analysis

The feasibility of our scheme has been demonstrated so far in terms of memory requirements. Let us consider lookup speed with memory-access timing. The on-chip process is straightforward. One on-chip engine running at 200 MHz can process one lookup in 5 ns, which is equivalent to 200 Mpps. As a contrast, off-chip memory access is a little more restricted and requires some analysis. Let us start with a quick review of current DRAM technology and discuss the timing analysis in detail.

Driven by an enormous market demand, DRAM speed and capacity are increasing rapidly while their power consumption and price are decreasing significantly [38], [39]. A state-of-the-art DRAM reaches 12.80 GB/s of throughput and 4 Gb of capacity [28], [29], and it has 8 banks on a chip. Using state-of-the-art DRAM technology (DDR3—1600 [Memory

TABLE V
NEXT-HOP INFORMATION MEMORY REQUIREMENTS (TAKING NEXT-HOP INFORMATION AS 8 BITS) COMPARED TO TREE BITMAP

	IPv4 (Real)		IPv4 (2M)		IPv6	
# of Routes	318,043		2,000,000		318,043	
	# of NHIsets	Memory	# of NHIsets	Memory	# of NHIsets	Memory
Tree Bitmap	315,462	2.4Mbits	1,999,580	15.3Mbits	318,036	2.4Mbits
PC-Trie 2	219,609	3.4Mbits	1,810,091	27.6Mbits	386,900	5.2Mbits
PC-Trie 4	153,936	4.7Mbits	1,514,739	46.2Mbits	361,773	11Mbits
PC-Trie 8	111,152	6.8Mbits	1,129,956	69Mbits	344,058	21Mbits
PC-Trie 16	80,790	9.9Mbits	715,048	87.3Mbits	321,410	39.2Mbits

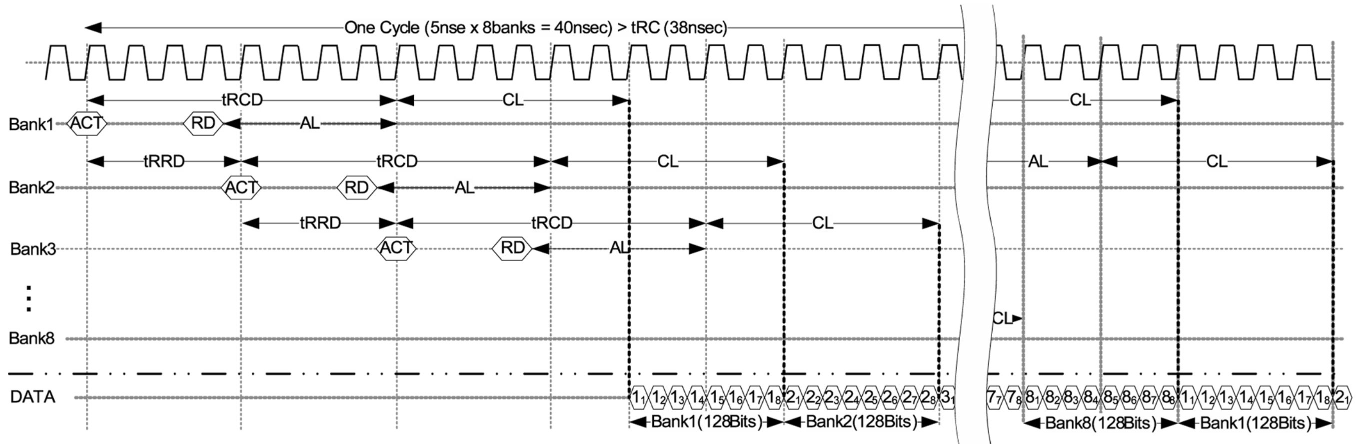

Fig. 13. DRAM memory access timing diagram ($\mathrm{tRCD} = 8, \mathrm{tRRD} = 4, \mathrm{CL} = 6, \mathrm{AL} = 5$).

clock 200 MHz, Bus clock 800 MHz]), only 5 ns is required to read 8 burst data (128 bits with 16-bit data bus). Thus, accessing 8 banks takes 40 ns, which satisfies timing restriction tRC (38 ns).

Fig. 13 illustrates how we continuously read 128 bits every 5 ns with a detailed timing diagram. Row activate (ACT) and read (RD) commands are sent sequentially following the timing specification of DDR3 SDRAM. For the sake of clear presentation, commands are presented for each bank. It is clear from the figure that data from each bank are output back to back after the $\mathrm{tRCD} + \mathrm{CL}$ cycle from ACT command.

By using on-chip membership query, FlashTrie needs only one PC-Trie memory access and one independent pipelineable NHI memory access. Hence, the minimum lookup time of the system reaches 5 ns per lookup, which is equivalent to 200 Mpps in the worst case. Therefore, two FlashTrie engines with two sets of DRAM chips (two PC-Trie and two NHI memories) can reach 400 Mpps. Worst-case IPv6 performance is 50% more than IPv4. The reason is that in FlashTrie, IPv4 and IPv6 have the same number of memory accesses per lookup. Even if we consider update time (10 Mpps per engine), lookup speed exceeds 100 Gb/s (250 Mpps). Moreover, the state-of-the-art FPGAs (Xilinx Virtex-6), which contain 38 Mb of block RAM, can have six engines in a single FPGA chip. With a single Virtex-6 FPGA and six sets of DDR3—1600 DRAM chips, FlashTrie can reach 1.2 Bpps, which is more than 480 Gb/s for IPv4 and more than 720 Gb/s for IPv6 in the worst case (for a minimum IPv6 packet size of 60 B). Note that our throughput is calculated based on the industry benchmark, where a packet is considered to be 50 B in IPv4 and 75 B in IPv6.

Tree Bitmap and FlashTrie timing analyses are presented in Fig. 14 with the IPv6 lookup example (considering that only the first 64 bits of IPv6 are used for lookup). The timing diagram

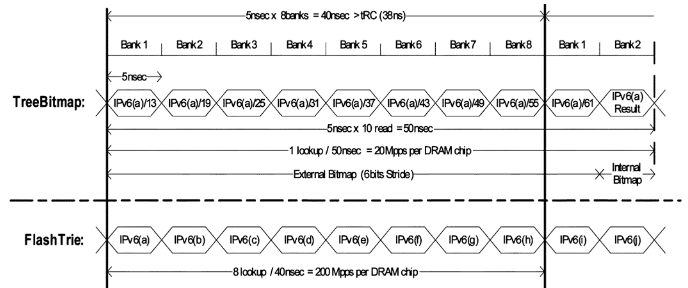

Fig. 14. DRAM memory access timing diagram for tree bitmap versus FlashTrie. DDR3—1600 DRAM is used for the comparison (5 ns $= 128$ bits read and $\mathrm{tRC} = 38$ ns).

compares the Tree Bitmap and FlashTrie schemes using the same resources. The Tree Bitmap uses the first 12 bits for direct lookup, and the rest of the bits are traversed in 6-bit strides. Assume that the Tree Bitmap uses optimization to fit the bitmap into one burst and ignores the DRAM read latency and processing time between levels. In this case, the Tree Bitmap approach requires 10 off-chip DRAM memory accesses (nine external bitmap and one internal bitmap accesses) for one lookup. Therefore, one IPv6 route lookup takes 50 ns, whereas FlashTrie can perform one lookup per 5 ns. Therefore, FlashTrie can finish 10 route lookups during the same period (50 ns). Hence, we can conclude that using the same number of memory chips, FlashTrie can perform 10 times faster compared to the Tree Bitmap. In other words, for the same throughput, Tree Bitmap needs 10 times more memory chips, which makes Tree Bitmap infeasible for high-speed route lookups.

VII. OPTIMIZATIONS

The amount of on-chip memory is always limited compared to off-chip memory. Efficient usage of on-chip memory leads

TABLE VI
ON-CHIP MEMORY REQUIREMENTS OF MULTIKEYS IN A BIN

	IPv4 (Real)			IPv4 (2M)			IPv6		
# of Keys	1	2	4	1	2	4	1	2	4
Direct Lookup	256 Kbits	256 Kbits	256 Kbits	256 Kbits	256 Kbits	256 Kbits	32 Kbits	32 Kbits	32 Kbits
Hash Table	0.32 Mbits	0.33 Mbits	0.34 Mbits	1.11 Mbits	1.15 Mbits	1.21 Mbits	2.1 Mbits	2.2 Mbits	2.3 Mbits
Hash ID Table	15 Kbits	7.5 Kbits	3.75 Kbits	60 Kbits	30 Kbits	15 Kbits	43 Kbits	21 Kbits	11 Kbits
Black Sheep Memory	0.53 Mbits	0.42 Mbits	0.34 Mbits	0.05 Mbits	0.02 Mbits	0 Mbits	6 Mbits	5.27 Mbits	4.67Mbits
Total	1.11 Mbits	1 Mbits -9.0%	0.93 Mbits -16.1%	1.47 Mbits	1.45 Mbits -1.4%	1.48 Mbits 0.7%	8.17 Mbits	7.53 Mbits -7.9%	7 Mbits -14.2%

us to achieve even higher throughput by duplicating multiple lookup engines. As we see in Section VI, Hash Table and Black Sheep memory consume the dominant portion of on-chip memory. In this section, we introduce two optimization methods that will reduce the memory requirements of these two modules.

A. Multiple Keys per Bin

This optimization contributes to reducing the BS memory requirement. The bin of the hash table takes two types of entries: one for the collision (overflow) and another for the noncollision (nonoverflow) cases as described in Section III-C. It contains root NHI and Verify Bits for a prefix in the nonoverflow case. Otherwise, it contains the BS memory address and the number of overflows. Assume a hash table allows only one key per bin, and one key is already programmed in the bin. If another key is assigned to the same bin, the bin is now used to store BS memory information (pointer to the BS and number of overflows). The key originally stored in the bin is relocated to the BS memory. Thus, the two keys are stored in the BS memory. It requires a total of three memory slots (one in the hash table and two in the BS memory), which means the memory overhead is 33.3%. Here, we apply multikey optimization, which can reduce the memory overhead caused by collisions (overflow). Allowing more than two keys per bin, we can eliminate the overhead of the overflow. According to our statistics, the majority of the bins has only one or two collisions. Thus taking this optimization, many bins will not require BS memory. This significantly reduces the BS memory requirements.

We experimented on two types of setting: allowing two or four keys per bin. The results are listed in Table VI. In the table, the multikey optimization is not applied to the first column for each routing table. The results of adopting two and four entries in a bin are compared to the first column in each routing table. The results show that the on-chip memory requirement of both IPv4 real routing table and IPv6 are decreased from 8% to 16%. Since the total number of entries remains the same, the reductions are mainly contributed from the BS memory, which means that fewer subtries are placed into BS memory. As a result, the density of hash table memory becomes higher.

B. Incremental Verify Bits

This optimization contributes to reducing the BS memory requirement as well as the hash table size in the membership query module. The hash-based scheme needs to perform a verification operation for the hash result to identify any false positive described in Section III-C. The Verify Bits are stored in the hash table and BS memories. Both memories are implemented in on-chip memory, so the size of the Verify Bits directly affects the memory requirements of on-chip memories. Without this optimization, the length of the Verify Bits increases proportionally to the length of the prefixes. For example, the size of the Verify Bits for IPv4/16 is 5 bits, and it becomes 16 bits for IPv4/25. Furthermore, IPv6 requires only 8 bits in the first level (IPv6/13), but increases the requirement to 49 bits in the bottom level (IPv6/58).

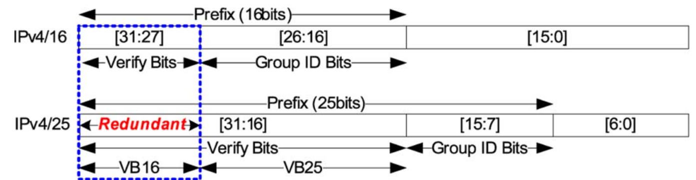

Fig. 15. Incremental Verify Bits optimization.

FlashTrie executes all levels of the membership query in parallel. Recall that all root NHIs of intermediate subtries (ancestors) exist in the membership query module as we mentioned in Section V. In this circumstance, a subtrie of the input IP address is matched in a lower level, which implies that the shorter subtries also exists in the upper levels. Let us take IPv4 as an example. If the subtrie of IPv4/25 exists, the root NHI of IPv4/16 can also be found in the membership query module. Verify Bits for both root of subtries are stored in the membership query module. As shown in Fig. 15, the Verify Bits of IPv4/16 consist of bits from 31 to 27 of the subtrie IP address, and Verify Bits of IPv4/25 consist of bits from 31 to 16 of the subtrie IP address. The Verify Bits [31:27] of the IPv4/25 and IPv4/16 are identical. Thus, the original scheme stores redundant Verify Bits in all levels. The membership query module of IPv4/16 stores Verify Bits of bits [31:27], marked as VB16 in the figure. The module of IPv4/25 needs only bits [26:16], marked as VB25. During the query process, the VB16 will be passed to the module of IPv4/25. The verification for the prefix of IPv4/25 will be compared with the concatenation of VB16 and VB25. In the case of IPv6, the storage of Verify Bits is organized as in Fig. 16. We call the scheme *Incremental Verify Bits*.

Table VII lists the on-chip memory requirements and comparisons of applying incremental Verify Bits optimization combined with multikeys optimization. However, the multikey optimization is not applied to the first column of each routing table. These columns show the performance of applying the incremental Verify Bits scheme only. Compared to the corresponding columns in Table VI, the scheme gains improvements. Even when multikeys optimization is applied, additional improvements remain achievable. Although the improvements for IPv4 are fractional, improvements in IPv6 is significant. The reason is that even though the requirements of hash table memory and BS

TABLE VII
ON-CHIP MEMORY REQUIREMENTS OF MULTIKEYS AND VERIFY BITS BYPASSING

	IPv4 (Real)			IPv4 (2M)			IPv6		
# of Keys	1	2	4	1	2	4	1	2	4
Direct Lookup	256 Kbits	256 Kbits	256 Kbits	256 Kbits	256 Kbits	256 Kbits	32 Kbits	32 Kbits	32 Kbits
Hash Table	0.3 Mbits	0.3 Mbits	0.31 Mbits	1.06 Mbits	1.09 Mbits	1.13 Mbits	0.93 Mbits	0.9 Mbits	0.89 Mbits
Hash ID Table	15 Kbits	7.5 Kbits	3.75 Kbits	60 Kbits	30 Kbits	15 Kbits	43 Kbits	21 Kbits	11 Kbits
Black Sheep Memory	0.52 Mbits	0.41 Mbits	0.33 Mbits	0.04 Mbits	0.02 Mbits	0 Mbits	3.67 Mbits	3.16 Mbits	2.76 Mbits
Total	1.08 Mbits	0.97 Mbits	0.9 Mbits	1.41 Mbits	1.38 Mbits	1.4 Mbits	4.67 Mbits	4.12 Mbits	3.69 Mbits
Comparison to Table VI	-2.8%	-3.4%	-3.9%	-3.5%	-4.4%	-5.3%	-42.8%	-45.3%	-47.3%

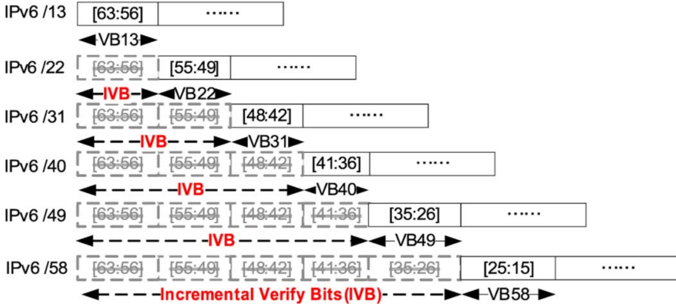

Fig. 16. Incremental Verify Bits optimization for IPv6.

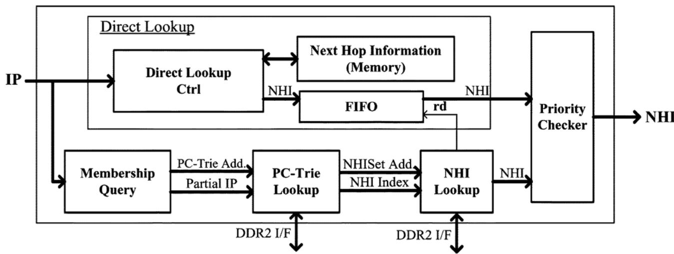

Fig. 17. Overview of hardware block diagram.

memory in IPv4/25 are decreased, their proportions are small. By this optimization, our result shows that over 40% of on-chip memory can be saved.

Applying these optimizations, the total on-chip memory requirements are reduced from 9.64 to 5.07 Mb (1.38 Mb—IPv4, 3.69 Mb—IPv6). With the state-of-the-art FPGAs that contain 38 Mb of dual-port on-chip memory, 14 lookup engines can be supported. As a result, a processing speed of 2.8 Gpps can be achieved.

VIII. HARDWARE IMPLEMENTATION

In this section, we describe detailed architecture of our prototype hardware implementation. The hardware utilization of the prototype design is also presented at the end of the section. Fig. 17 shows block diagrams of the FlashTrie lookup engine. The engine consists of five main submodules: *Direct Lookup*, *Membership Query*, *PC-Trie Lookup*, *NHI Lookup*, and *Priority Checker*. When performing lookup, the input IP address is fed into the Direct Lookup and Membership Query modules. The Direct Lookup module resolves the NHI in level zero, while the others cope with levels one and two. The Membership Query module checks if there are existent subtries for the input IP address in the next two levels. If there are, the module generates a PC-Trie address to query off-chip memory, where the bitmap and memory address of NHI set are stored, and forwards it to the PC-Trie Lookup module. The PC-Trie Lookup module is responsible for reading the data from external memory in terms of the input address, traversing the bitmap for further prefix matching, and calculating the address of the target NHI set. Then, the NHI Lookup module uses the address to obtain a set of NHIs from another off-chip memory and picks out the final one with partial IP according to the FlashTrie algorithm.

Each IP address query has an exact match in level zero, and it also may have subtrie matches in levels one and two. The outcome from the longest one will be the final NHI choice. At the end, the Priority Checker module selects the NHI from Direct Lookup module as output if only a subtrie matches in the level zero; otherwise, it returns the NHI from the NHI Lookup module.

A. Hardware Implementation of Direct Lookup Module

The Direct Lookup module is used for level-zero NHI lookup. The NHI for this level is stored in an on-chip Block RAM. The module consults the on-chip Block RAM according to the input IP address. Here, only most- significant bits from 31 to 17 are used as the indexing address to fetch NHI out of memory. The fetched NHI is stored in a first-in–first-out (FIFO) buffer until matching procedures in other levels are completed. Then, the NHI for the input IP address is chosen between them.

B. Hardware Implementation of Membership Query Module

Once the IP address is input into the lookup engine, the subtrie matching procedures of variant length are executed in parallel. The membership query module checks if subtries exist for the input IP address. For the IPv4 lookup, the binary trie is divided into levels of prefixes of lengths 16 and 25 and marked as level one and two, respectively. As shown in Fig. 18, each level has an individual PC-Trie Address Generator for parallel lookup and PC-Trie address generation. Bits [31:16] of the input IP address are used as the input for module /16, while bits [31:7] are for module /25. Bits [15:0] of the IP address are stored in the FIFO buffer for further use. The generators for different levels are basically the same in structure and operations. The generator for level two is used as an example here and shown in Fig. 19. To store plenty of subtries in limited memory slots, the HashTune is adopted. The subtries are grouped in terms of partial bits of its prefix. Each group can use a hash function to determine the position of the memory slot for storing the information of the subtrie. The LSB 11 bits of the generator's input are treated as a group number to query the Hash ID memory for its hash function. Then, the whole input is involved in calculating the address, used to query the on-chip hash table memory, by employing HashTune. The output from hash table memory can be interpreted in two ways. As mentioned in Section III, it is possible that more than one subtrie is hashed into the same bin, which results in collision. In this case, the output is taken as a query address to further refer to the BS memory. Otherwise, it is

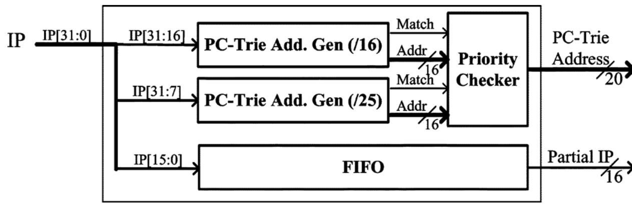

Fig. 18. Membership Query module.

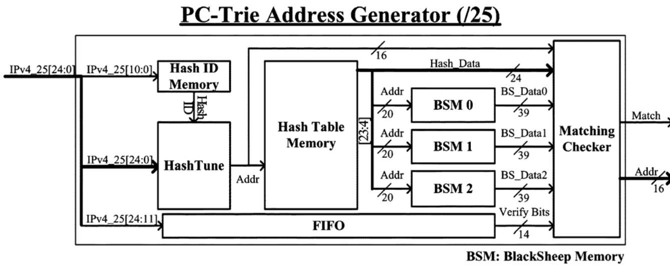

Fig. 19. Detail of the PC-Trie address generator.

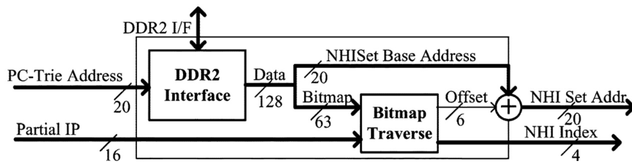

Fig. 20. Block diagram of PC-Trie processing unit.

used to verify the existence of the subtrie for the input by comparing the Verify Bits from the output with the input. In the actual implementation, the BS memories are queried whether the collision happens or not. Then, the verification compares the bits of the input with the outcomes from all BS memories as well as the hash table memory in the Matching Checker module. If it is identical to the one from hash table, the output address of the Matching Checker module is the same as the one used to query the hash table. On the contrary, it will be extracted from the BS memory's outcome, whichever Verify Bits of the BS memory entry are matched with the input IP address.

C. Hardware Implementation of PC-Trie and NHI Lookup

If a subtrie is found for the input IP address in the Membership Query module, the IP address may have longer prefix matching in the subtrie, which is stored as a PC-Trie. The PC-Trie Lookup module, as shown in Fig. 20, uses the address generated by the membership query module, to refer to the off-chip memory. The DDR2 interface employs a DDR2 memory controller, generated by the Xilinx Memory Interface Generator (MIG). The data from the memory consists of the bitmap and NHI set base address for a subtrie. The bitmap is traversed to determine further matching with the partial IP address, the remainder from the membership query stage. Then, the offset of the memory position is counted as mentioned in Section III and is added to the NHI set base address, resulting in the address of the NHI set. The partial IP address results in the NHI index by removing the bits used for partial matching. Then, the NHI lookup module, whose architecture is shown in Fig. 21, applies the NHI set address to query another off-chip memory, which stores the NHI set data. The resulting NHI set contains 16 NHIs. The final one is selected based on the NHI index.

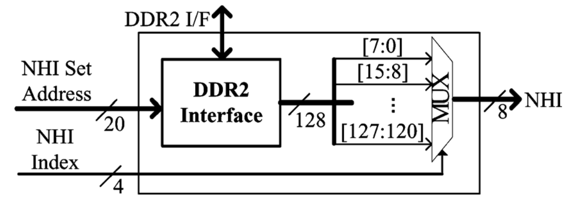

Fig. 21. Block diagram of NHI processing unit.

TABLE VIII
HARDWARE RESOURCE REQUIREMENTS

	Slices	Slice Reg.	LUTs	LUT RAM	Block RAM/FIFO
Direct Lookup	46	52	68	0	17
MemberShip Query	683	335	836	1	87
PC-Trie Lookup	1,677	1,578	1,615	109	2
NHI Lookup	1,457	1,316	1,325	75	2
Others	720	190	360	21	0
Total	4,583	3,471	4,204	206	108

D. Hardware Implementation Result

The FlashTrie lookup engine is implemented on a Xilinx Virtex-4 FX100 FPGA chip, which has 42 176 slices and 376 block RAMs. The development environment is ISE 10.1. The design is downloaded on the FPGA on a board PLDA XpressFX. The board is equipped with two 1-Gb DDR2 SDRAMs. Each independently connects to the FPGA chip. The prototype design uses one DRAM chip for bitmap storage and another is for NHI. Both DRAMs have a 16-bit data bus. The burst size is configured as 8, in which 128 bits of data can be read once. The whole design, including the DRAMs, is running at 200 MHz (bus clock 400 MHz). Under this clock frequency, 200 Mpps per engine can be achieved. The memory controllers are generated by the Xilinx Memory Interface Generator (MIG 2.3). The input to the engine is placed in an on-chip memory and fed into it. The resource utilizations are listed in Table VIII. One engine is employed in the prototype. The total occupied slices are 8%, and Lookup tables (LUTs) are 4%. The used block RAM/FIFOs are 29% of the FPGA capacity.

IX. CONCLUSION

In this paper, we proposed a low-cost, high-speed, next-generation route lookup architecture called FlashTrie that can support 2 M IPv4 and 318 k IPv6 routes simultaneously. A new compact data structure for a multibit-trie representation, called Prefix-Compressed Trie, and a hash-based, high-speed, memory-efficient architecture are proposed. Comprehensive simulation results and hardware implementation show the FlashTrie architecture can achieve 160-Gb/s worst-case throughput with four DDR3 DRAM chips. This exceptionally small number of off-chip memory requires very little I/O pins in the main control chips and maintains low system cost. FlashTrie can support real-time incremental updates by reserving only 5% of total throughput (10 Mpps per engine).

REFERENCES

[1] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software IP lookups with incremental updates," *Comput Commun Rev*, vol. 34, no. 2, pp. 97–122, 2004.

[2] Juniper Networks, “Industry leaders demonstrate 100 Gigabit Ethernet interoperability at OFC,” Press Release, 2010 [Online]. Available: http://www.juniper.net
[3] Cisco, “Cisco introduces foundation for next-generation Internet: The Cisco CRS-3 carrier routing system,” Cisco Press Release, 2010 [Online]. Available: http://newsroom.cisco.com
[4] IEEE 802.3 Ethernet Working Group, “IEEE P802.3ba 40 Gb/s and 100 Gb/s Ethernet Task Force,” 2010 [Online]. Available: http://www.ieee802.org/3/ba/
[5] Cisco, “Approaching the zettabyte era,” Cisco White Paper, 2008 [Online]. Available: http://www.cisco.com
[6] T. Bates, P. Smith, and G. Huston, “CIDR report,” [Online]. Available: http://www.cidr-report.org/
[7] F. Zane, G. Narlikar, and A. Basu, “CoolCAMs: Power-efficient TCAMs for forwarding engines,” in *Proc. IEEE INFOCOM*, 2003, pp. 42–52.
[8] K. Zheng, C. Hu, H. Lu, and B. Liu, “A TCAM-based distributed parallel IP lookup scheme and performance analysis,” *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 863–875, Aug. 2006.
[9] P. Gupta, S. Lin, and N. McKeown, “Routing lookups in hardware at memory access speeds,” in *Proc. IEEE INFOCOM*, 1998, pp. 1240–1247.
[10] N.-F. Huang and S.-M. Zhao, “A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers,” *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1093–1104, Jun. 1999.
[11] N.-F. Huang, S.-M. Zhao, J.-Y. Pan, and C.-A. Su, “A fast IP routing lookup scheme for gigabit switching routers,” in *Proc. IEEE INFOCOM*, 1999, vol. 3, pp. 1429–1436.
[12] V. Srinivasan and G. Varghese, “Fast address lookups using controlled prefix expansion,” *Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, 1999.
[13] S. Cadambi, S. Chakradhar, and H. Shibata, “Prefix processing technique for faster IP routing,” US Patent 2006/0 200 581 A1, 7 398 278, 2005.
[14] S. Kaxiras and G. Keramidas, “IPStash: A set-associative memory approach for efficient IP-lookup,” in *Proc. IEEE INFOCOM*, 2005, vol. 2, pp. 992–1001.
[15] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, “Chisel: A storage-efficient, collision-free hash-based network processing architecture,” in *Proc. ISCA*, 2006, pp. 203–215.
[16] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, “Fast hash table lookup using extended bloom filter: An aid to network processing,” in *Proc. ACM SIGCOMM*, 2005, pp. 181–192.
[17] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, “Longest prefix matching using bloom filters,” *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 397–409, Apr. 2006.
[18] H. Song, F. Hao, M. Kodialam, and T. Lakshman, “IPv6 lookups using distributed and load balanced bloom filters for 100 Gbps core router line cards,” in *Proc. IEEE INFOCOM*, 2009, pp. 2518–2526.
[19] M. Bando, N. S. Artan, and H. J. Chao, “FlashLook: 100 Gbps hash-tuned route lookup architecture,” in *Proc. HPSR*, 2009.
[20] M. Hanna, S. Demetriades, S. Cho, and R. Melhem, “Progressive hashing for packet processing using set associative memory,” in *Proc. ACM/IEEE ANCS*, 2009, pp. 153–162.
[21] S. Sikka and G. Varghese, “Memory-efficient state lookups with fast updates,” in *Proc. ACM SIGCOMM*, 2000, pp. 335–347.
[22] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, “Scalable, memory efficient, high-speed IP lookup algorithms,” *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 802–812, Aug. 2005.
[23] H. Song, J. Turner, and J. Lockwood, “Shape shifting tries for faster IP route lookup,” in *Proc. IEEE ICNP*, 2005, pp. 358–367.
[24] A. Basu and G. Narlikar, “Fast incremental updates for pipelined forwarding engines,” *IEEE/ACM Trans. Netw.*, vol. 13, no. 3, pp. 690–703, Jun. 2005.
[25] W. Jiang and V. K. Prasanna, “Multi-terabit IP lookup using parallel bidirectional pipelines,” in *Proc. CF*, 2008, pp. 241–250.
[26] H. Le and V. Prasanna, “Scalable high throughput and power efficient IP-lookup on FPGA,” in *Proc. IEEE FCCM*, 2009, pp. 167–174.
[27] S. Kumar, M. Becchi, P. Crowley, and J. Turner, “CAMP: Fast and efficient IP lookup architecture,” in *Proc. ACM/IEEE ANCS*, 2006, pp. 51–60.
[28] Samsung, “Samsung DRAM,” [Online]. Available: http://www.samsung.com/
[29] Micron, “Micron DRAM,” [Online]. Available: http://www.micron.com/
[30] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[31] S. Kumar and P. Crowley, “Segmented hash: An efficient hash table implementation for high performance networking subsystems,” in *Proc. ACM ANCS*, 2005, pp. 91–103.
[32] A. Kirsch and M. Mitzenmacher, “The power of one move: Hashing schemes for hardware,” in *Proc. IEEE INFOCOM*, 2008, pp. 106–110.
[33] N. Artan, H. Yuan, and H. Chao, “A dynamic load-balanced hashing scheme for networking applications,” in *Proc. IEEE GLOBECOM*, 2008, pp. 1–6.
[34] M. Mitzenmacher, “The power of two choices in randomized load balancing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001.
[35] F. Hao, M. Kodialam, and T. V. Lakshman, “Building high accuracy bloom filters using partitioned hashing,” in *Proc. ACM SIGMETRICS*, 2007, pp. 277–288.
[36] University of Oregon, “University of Oregon Route Views Project,” 2005 [Online]. Available: http://www.routeviews.org
[37] M. Wang, S. Deering, T. Hain, and L. Dunn, “Non-random generator for IPv6 tables,” in *Proc. IEEE HOTI*, 2004, pp. 35–40.
[38] D. Klein, “Memory technology trends and challenges,” 2005 [Online]. Available: http://www.jedex.org/images/pdf/d_klein_keynote.pdf
[39] R. Advani, “Server memory solutions for the impending data center power crisis,” 2008 [Online]. Available: http://download.micron.com/pdf/whitepapers/server_memory_white_paper_hi.pdf

Masanori Bando (A’10) received the B.S. and M.S. degrees in electrical engineering from Tamagawa University, Tokyo, Japan, in 1999 and 2001, respectively, and is currently pursuing the Ph.D. degree, working on high-speed network security and next-generation router architectures at the High Speed Network Lab, Polytechnic Institute of New York University, Brooklyn.

Yi-Li Lin received the B.S. and M.S. degrees in computer science and information engineering from National Cheng Kung University (NCKU), Tainan, Taiwan, in 2002 and 2004, respectively, and is currently pursuing the Ph.D. degree in the computer science and information engineering at NCKU.

He was an exchange student, from November 2009 to May 2010, with the Department of Electrical and Computer Engineering, Polytechnic Institute of New York University, Brooklyn. His research interests include video codec, VLSI design, Electronic System Level (ESL) design methodology/tools, and FPGA acceleration for VLSI verification.

H. Jonathan Chao (S’82–M’83–SM’95–F’01) received the Ph.D. degree in electrical engineering from The Ohio State University, Columbus, in 1985.

He is the Department Head and a Professor of electrical and computer engineering with the Polytechnic Institute of New York University, Brooklyn, which he joined in January 1992. During 2000 to 2001, he was Co-Founder and CTO of Coree Networks, Tinton Falls, NJ, where he led a team to implement a multiterabit multi-protocol label switching (MPLS) switch router with carrier-class reliability. From 1985 to 1992, he was a Member of Technical Staff with Telcordia, Piscatway, NJ. He holds 42 patents with 10 pending and has published over 200 journal and conference papers. He coauthored three networking books, *Broadband Packet Switching Technologies* (Wiley, 2001), , *Quality of Service Control in High-Speed Networks* (Wiley, 2001), and *High-Performance Switches and Routers* (Wiley, 2007). He has also served as a consultant for various companies, such as Huawei, Lucent, NEC, and Telcordia.. He has been doing research in the areas of network designs in data centers, terabit switches/routers, network security, network on the chip, and biomedical devices.

Prof. Chao is a Fellow of the IEEE for his contributions to the architecture and application of VLSI circuits in high-speed packet networks. He received the Telcordia Excellence Award in 1987