This paper was presented as part of the Mini-Conference at IEEE INFOCOM 2011

Approximately-Perfect Hashing: Improving Network Throughput through Efficient Off-chip Routing Table Lookup

Zhuo Huang, Jih-Kwon Peir, Shigang Chen
Department of Computer & Information Science & Engineering, University of Florida
Gainesville, FL, 32611, USA
{zhuang, peir, sgchen }@cise.ufl.edu

***Abstract*—IP lookup is one of the key functions in the design of core routers. Its efficiency determines how fast a router can forward packets. As new content is continuously brought to the Internet, novel routing technologies must be developed to meet the increasing throughput demand. Hash-based lookup schemes are promising because they have low lookup delays and can handle large routing tables. To achieve high throughput, we must choose the hash function to reduce the lookup bandwidth from the off-chip memory where the routing table is stored. The routing table updates also need to be handled to avoid costly re-setup. In this paper, we propose AP-Hash, an approximately perfect hashing approach that not only distributes routing-table entries evenly in the hash buckets but also handles routing table updates with low overhead. We also present an enhanced approach, called AP-Hash-E, which is able to process far more updates before a complete re-setup becomes necessary. Experimental results based on real routing tables show that our new hashing approaches achieve a throughput of 250M packets per second and perform re-setup as few as just once per month.**

I. Introduction

The Internet is becoming the world's communication backbone as more and more content is moving from the traditional media into the IP domain. Network throughput is ever increasing, driven by bandwidth demand from new applications, including IPTV, peer-to-peer video streaming, online games, cloud computing, E-commerce, distrusted data center network, etc. The line speed of a core router is expected to reach terabits per second in near future. To match the optical speed, the routing-table lookup function must also be upgraded.

Each entry in a routing table consists of an address prefix (or prefix in short), output port and other routing information. When a router receives a packet, it extracts the destination address, matches it against the prefixes in the routing table, and forwards the packet based on the routing information in the matching entry. If there are multiple matches, the entry with the longest prefix wins. This is called the *longest prefix match (LPM)*.

The routing tables on todays core routers contain more than 300K prefixes and their sizes are continually growing [1]. Routing tables for the future IPv6 will be much larger. Although the routers have fast on-chip memory, the size of such memory is usually insufficient to hold a large routing table. Hence, the routing table is likely stored in off-chip memory. During IP lookup, routing information is fetched from off-chip memory, which is the most time-consuming step and likely bottleneck that determines the throughput. It is essential to optimize off-chip access and allow incrementally routing table updates without frequent re-setup of the whole table.

Existing routing-table lookup schemes can be classified into three categories, which are TCAM-based schemes [2], trie-based schemes [3] and hash-based schemes [4], [5]. However, ever-increasing routing table sizes and speed demands bring severe challenges to the TCAM-based and tried-based schemes. The TCAM-based schemes can only handle the small or middle-sized routing tables due to high hardware and power costs. The trie-based schemes need multiple memory accesses per packet and have variable routing time.

Hash-based lookup schemes have low lookup delays and can handle a large routing table. The routing table is stored in an array of off-chip hash buckets. Each bucket contains a small number of prefixes and the associated routing information. The bucket in which a specific prefix is stored is determined by one or multiple hash functions (using the prefix as input). When a router receives a packet, it performs two tasks. First, it determines the size of the prefix it should extract from the destination address for longest prefix match in routing-table lookup. Second, it determines which bucket(s) should be fetched from off-chip memory in order to find a matching prefix.

The first task has been solved by Controlled Prefix Expansion [6] or Bloom filter and its variances [7], [4]. The focus of this paper is on the second task. Once an appropriate prefix is extracted from the destination address, the router has to use this prefix to identify and fetch one or multiple hash buckets to the processor chip, which will then find the matching routing entry. To maximize the throughput, we want to minimize the amount of data that is fetched off-chip per lookup.

The bucket size is determined by the largest number of prefixes (together with the associated routing information) that a bucket has to store. Balanced distribution of prefixes in the hash buckets help reduce the bucket size. Using a single hash function will result in a large bucket size and high off-chip overhead since it cannot achieve balanced distribution. Multiple Hash functions can be used to balance the prefix distribution in the buckets [8], [9] but those schemes need to fetch multiple buckets per lookup. Perfect,

978-1-4244-9921-2/11/$26.00 ©2011 IEEE

TABLE I
NOTATIONS

Symbol	Meaning
N	Set of keys (prefixes)
n	Number of keys
M	Set of buckets
m	Number of buckets
b	Bucket size
d	Number of Hash functions
s	Bits of one prefix and its routing information
B	Bandwidth in terms of number of bits per second
L	Set of entries in HIT
l	Number of entries in HIT
h_i	i-th hash function that maps prefix to the hash table
g	Hash functions that maps prefix to HIT
$HIT(i)$	i-th entry in HIT
G_i	A group of prefixes x, where $g(x) = i$

collision-free hash functions [10] guarantee the minimum bucket size. However, they can only work with a pre-determined prefix set and require considerable on-chip space to record the hash function itself [11].

In this paper, we propose a new hashing approach called *Approximately-Perfect Hashing (AP-Hash)*. It uses a small number of hash functions and a small on-chip *Hash Index Table (HIT)* to balance the prefix distribution in the hash buckets. The AP-hash performs two levels of hashing and table access: on-chip access to the HIT, and off-chip access to the routing table that is stored in hash buckets. During the setup, we first hash each prefix to an HIT entry, which stores a hash function index, indicating which specific hashing function we shall use to place the prefix in the hash buckets. By adjusting the values in the HIT, we can alter the placement of the prefixes in the buckets and achieve almost perfect distribution. During the lookup of a prefix, we hash the prefix to an HIT entry, retrieve the hash function index, and use the corresponding hash function to identify an off-chip bucket for retrieval. The AP-Hash can handle routing-table updates with minimum HIT re-setups. When new prefixes are inserted in the routing table, one or more HIT entries are modified to avoid overflowing any bucket.

This paper makes several key contributions. First, to the best of our knowledge, the AP-Hash is the first that uses a small intermediate table to record the hashing function for each address prefix in order to balance the prefix distribution in a hash-based routing table. Second, the AP-Hash provides an efficient solution to handle frequent routing-table updates to minimize the costly re-setup. Third, we evaluate the AP-Hash and its enhancement AP-Hash-E using real routing tables from the core routers. Experiments show that they can deliver a lookup rate of 250 million packets per second to accommodate over 100 Gbps high-speed core router line cards. In addition, based on the routing table update traces, we find that the AP-Hash-E only performs the routing table re-setup once a month.

II. Existing Hash Approaches

In this section, we review the existing hashing-based routing schemes. Suppose n prefixes are mapped through hash functions to m buckets for storage. The bucket size is determined by the largest number of prefixes (and their routing information) that have to be stored in any bucket. When a packet arrives, the router first determines the length of the longest-matched prefix l_p as described in [5]. Then, the first l_p bits of the destination address is hashed to locate the bucket in the hash table. The whole bucket is fetched to a network processor chip, where all prefixes in the bucket are compared with the destination address in parallel to find the matching prefix and its routing information.

The IP-lookup procedure on a router can be pipelined to overlap the different stages such as determining the prefix length, fetching buckets from off-chip, and prefix comparison. The most time-consuming step is fetching buckets from off-chip, which determines maximum throughput. Let B be the maximum bandwidth between the network processor chip and the memory, s be the size of a routing-table entry, including the prefix and its routing information, b be the bucket size, i.e., the number of prefixes that a bucket can store, and d be the number of buckets the router has to fetch for each lookup. The maximum routing throughput is $\frac{B}{dbs}$. Notations are given in Table I.

Since B and s are usually pre-determined, we should minimize both d and b to maximize the routing throughput $\frac{B}{dbs}$. The optimal routing throughput is $\frac{Bm}{sn}$ where $d = 1$ and $b = \frac{n}{m}$. Hence, an ideal hash-based lookup scheme will fetch a single bucket per lookup and evenly distribute prefixes in hash buckets such that b is minimized to $\frac{n}{m}$. However, the existing hashing approaches cannot achieve these goals.

Although approaches using a single general hash function only need to fetch one bucket, the bucket size is usually large due to the unbalance among the buckets. It is well known that multiple hash functions can be used to reduce the bucket size [13]. Suppose we use d hash functions. Each prefix is hashed to d buckets and the prefix is placed into the bucket currently having the fewest prefixes. In general, using multiple hash functions can achieve a near-optimal bucket size, but it requires fetching d buckets during lookup. Therefore, the throughput is limited to $\frac{Bm}{dsn}$ which is only $\frac{1}{d}$ of the optimal.

Perfect hash function, which can guarantee the minimum bucket size, has been applied in routing table lookup problem [10]. Although the bucket size is reduced to the minimal, any change to the routing table requires updating the perfect hash function itself, which is a costly operation. Moreover, the current best approach still requires more than $2n$ bits to encode the minimal perfect hash function [11].

III. Approximately-Perfect Hash function AP-Hash

A. Basic Design

The AP-Hash uses d hash functions so that each prefix can be placed into one of the d buckets to balance the buckets. In this approach, instead of fetching all d buckets, it fetches only a single bucket for each routing lookup. To accomplish this, the AP-Hash uses a small *Hash Index Table (HIT)* to record the id of the hash function which each prefix used. The HIT has l entries and each entry contains e bits to encode the d choices of the hash functions, where $e = \log d$. As shown in Figure 1, each prefix is first hashed to an entry in HIT to select the predetermined hashing function for the prefix.

Note that the AP-Hash can work with any value of d and l. In our study, we narrow the value of d to be a power of 2 to make full use of the encoded bits in each HIT entry. We also limit the number of hashing functions to 8 to confine

TABLE II
SINGLE HASH, D-LEFT HASH, PERFECT HASH AND AP-HASH

	bucket size b	bits fetched per lookup	on-chip space (bits)	off-chip space (bits)	update
Single Hash	$\Theta(\frac{\ln m}{\ln\lceil 1+\frac{m}{n}\ln n\rceil}+\frac{n}{m})$[12]	$b\times s$	0	$b\times m\times s$	straightforward
Multiple Hash	$(1+\circ(1))\times\frac{\ln\ln m}{\ln d}+\Omega(\frac{n}{m})$[13]	$b\times s\times d$	0	$b\times m\times s$	straightforward
Perfect Hash	$\frac{n}{m}$	$b\times s$	$>2n$	$b\times m\times s$	re-setup involved
AP-Hash	$\sim\frac{n}{m}$	$b\times s$	$l\times l$	$b\times m\times s$	low cost

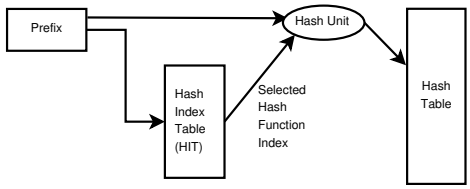

Fig. 1. AP-Hash Diagram

the space overhead. In addition, the HIT size is limited to $l=\frac{n}{2}$ entries, such that the on-chip space requirement is $0.5n\sim 1.5n$ bits when 2-8 hash functions are used. Note also that the extra HIT access increases the lookup time. However, due to its small size and on-chip implementation, this additional delay can be hidden in the pipelined design.

In a network, a router needs to handle the setup, lookup, and update of the routing table. In the AP-Hash, the setup stage establishes the HIT for the selected hashing functions. The hashed routing table can be arranged accordingly. During the lookup stage, a hashing function is selected from the HIT, and the hashed bucket is fetched from off-chip memory. The bucket size determines the achievable network throughput. Finally, the AP-Hash must modify the HIT and the routing table when network configurations are changed. These important functions are described in the following subsections.

B. AP-Hash Setup

In HIT setup, a complete search of all d^l combinations for the best HIT values is unrealistic. Instead, we propose a two-step greedy setup approach including the first-setup and the refine-setup.

In first-setup, we begin with an empty HIT. First, a general hash function $g(x)$ is selected. Then, the key set N (i.e., prefixes in the routing table) is divided to l subgroups, $G_0,\ G_1,\ \cdots,\ G_{l-1}$, where $G_i=\{x|x\in N\ and\ g(x)=i\}$. AP-Hash hashes all the prefixes in subgroup G_i into the buckets using the same hash function $h_{HIT(i)}$.

In determining the best hash function $h_{HIT(i)}$, a data structure named *bucket-load counter* is introduced to count and compare the loads for all buckets. The bucket-load counter is defined as $blc=[b,v_b,v_{b-1},v_{b-2},\cdots,v_0]$ while b is the largest bucket size and v_j is the number of buckets that contains j keys (prefixes). Assume $blc=[b,v_b,v_{b-1},v_{b-2},\cdots,v_0]$ and $blc'=[b',v'_b,v'_{b-1},v'_{b-2},\cdots,v'_0]$ are two bucket-load counters. We define blc is smaller than blc' ($blc<blc'$) if one of the following conditions is true.1) $b<b'$; 2) $b=b'$ and $v_b<v'_b$; 3) $b=b'$ and $v_i=v'_i$ $(t<i\le b)$ and $v_t<v'_t$.

We start with G_0, select the hash function h_j among all d hash functions that results in the smallest bucket-load counter, set $HIT(0)=j$, and use the hash function h_j to hash all prefixes in G_0 to the buckets. $G_1,G_2,\cdots,G_{l-1}$ are consequently processed. The first-setup algorithm is summarized in Figure 2.

```
for i in [0, x − 1] do
    Calculate G_i
    for each j, 0 ≤ j ≤ d − 1 do
        for each prefix p ∈ G_i do
            Calculate h_j(p)
            Assign key p to the bucket h_j(p)
        end for
        Calculate the bucket-load vector blv_j
    end for
    Find the min-max vector blv_j and set V(i) = j
    Assign all the keys in G_i to buckets using h_j
end for
```

Fig. 2. AP-Hash First-Setup

The first-setup may not produce an approximately perfect bucket size. The main reason is that only the prefixes in $G_0,G_1,\cdots,G_i$ are involved in the load computation when setting up $HIT(i)$, but the prefixes in $G_{i+1},\cdots,G_{l-1}$ are not considered. The refine-setup adjusts the HIT values. It first retries all d possible values for $HIV(0)$ and moves the prefixes in G_0 to the new buckets. If any hash function results in a smaller bucket-load counter than the existing one, the $HIT(0)$ is updated to use the new hash function. This procedure repeats for $HIV(1),\cdots,\ HIV(l-1)$. The refine-setup procedure can be iteratively performed multiple times. However, one-time Refine-Setup usually produces an approximately perfect bucket size.

C. Hash Table Update

There are three types of updates to a hash table, 1) inserting new prefixes, 2) deleting existing prefixes, and 3) modifying the routing information of an existing prefix. The deletion and modification for the AP-Hash are straightforward. It searches the prefix to be deleted or modified and removes it from the bucket or make any necessary modification. However, inserting new prefixes is more complicated. Although we can decide which bucket the new prefix should be placed by finding the hashing function from the HIT, the hashed bucket for the new prefix may already be full. Inserting new prefixes into a full bucket will prevent the bucket from being fetched in time to sustain the target network throughput.

In this case, a straightforward solution is to re-setup the whole HIT. However, re-setup is costly and not suitable for frequent prefix insertions in the routing table. One acceptable solution is to mildly modify the HIT and the routing table when inserting a new prefix. However, adjusting an HIT entry may affect multiple prefixes that have already been placed into buckets. We add another data structure, *HIT mapping table*, to handle the updates. The HIT mapping table records each prefix subgroup G_i, $0\le i\le l-1$. It is located in the off-chip memory and only accessed when a prefix is inserted or deleted.

The procedure to insert a prefix x works as follows. First, we calculate $i=g(x)$, put x into G_i in the HIT mapping table and calculate $h(x)=h_{HIT(i)}(x)$. If the bucket $B(h(x))$ still has room to hold the new prefix,

the new prefix is inserted to the bucket and the insertion finishes. Otherwise, we try to modify $HIT(i)$ to fit all prefixes. The prefix subgroup G_i is fetched from the HIT mapping table. We select a new hashing function h_j and check whether the new prefix x and all the prefixes in G_i can find a room in the new buckets when using h_j . If successful, we set $HIT(i) = j$ and move all the prefixes in G_i to the new buckets. Otherwise, we try another hash function h_j. If all the possible hashing functions fail, the HIT re-setup procedure is invoked.

D. Ap-Hash-E: an Enhancement of AP-Hash

In this subsection, we propose an enhanced version of AP-Hash, named AP-Hash-E, which modifies multiple HIT indices to resolve the bucket overflow problem. AP-Hash fails to insert a new prefix if all values of the HIT entry which the new prefix is hashed to result in prefix overflow in one or more buckets. In this case, instead of resetting all the entries of the HIT, the new AP-Hash-E tries to remove some prefix in the overflow bucket to other buckets in hope to avoid the overflow problem.

The AP-Hash-E works as follows. During the AP-Hash insertion procedure, the prefix subgroup G_i is fetched from the HIT mapping table. We select a new hashing function h_j and check whether the new x and all the prefixes in G_i can find a room in the new buckets indexed by using h_j . When h_j fails due to overflow, the hashing function h_j is remembered if it produces a *minimum* overflow. A minimum overflow produces a single prefix overflow in only one bucket. When all hashing functions of $HIT(i)$ fails, the new AP-Hash-E tries to resolve the detected minimum overflow cases one-by-one.

In resolving a minimum overflow, it selects one prefix in the overflowed bucket, removes it from the bucket, treats it as a new prefix to insert and calls the AP-Hash algorithm presented earlier. The AP-Hash-E is successful whenever the overflow is resolved. The prefix in G_i, which is just placed into the overflowed bucket, will not be selected. The procedure repeats until the overflow is resolved or all the prefixes in the overflowed bucket are tried. A re-setup procedure is only invoked when all the detected minimum overflow cases cannot be resolved.

IV. Performance Evaluation

When a packet arrives, the router first determines the length of the longest-matched prefix l_p as described in [7], [4], [5]. Then, the first l_p bits of the destination address plus its length are used to hash the bucket in the hash table. In the AP-Hash, the l_p bits and its length are first hashed to the HIT using a general hashing function in [14]. The selected hashing function is then used to hash the l_p bits plus its length to locate the bucket in the hash table.

Five routing tables from the Internet backbone routers are used in our experiments: as286, as513, as1103, as4608, and as4777, downloaded from [1]. These tables are dumped from the routers at 7:59am January 1st, 2010 and contain 276K, 291K, 279K, 283K, 281K prefixes respectively.

We compare three hashing schemes in our experiments: *single-hash*, *d-left-hash* (d=2 and d=4) [4]; and the *AP-hash* ($e = 2$, $l = \frac{n}{2}$), which is proposed in this paper using the hash functions in [14]. In each experiment, we vary

TABLE III
Throughput, memory sizes for three hash schemes with $m = 350,000$

	bucket size	Throughput (M/sec)	bits pre key on-chip	bits pre key off-chip
Single Hash	7.62	65.61	0	604.4
2-left Hash	3	83.3	0	237.9
4-left Hash	2	62.5	0	159.6
AP-Hash	2	**250**	1	223.6

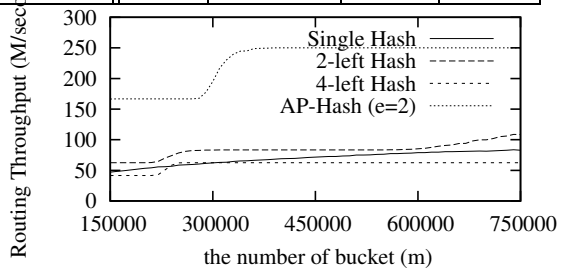

Fig. 3. The average routing throughput of 5 routing tables for Single Hash,, d-left ($d = 2$ and $d = 4$) and AP-Hash, under different number of buckets.

the number of buckets, m between 150K and 750K. We try all five routing tables for each studied hashing scheme and show the average result of all five tables based on the average of 100 simulations.

Table III summaries the experiment results of the three hash schemes when $m = 350,000$. We compare the bucket size, the routing throughput, and the memory size for the three simulated schemes. We assume each prefix and its routing information are stored in 64 bits. We use the state-of-art QDRTMIII SRAM to evaluate the routing throughput. The current QDR-III SRAM runs at 500MHz and supports 64- bit read/write operations per cycle. The throughput can be calculated as $\frac{500}{b}$ for single hash and AP-Hash, and $\frac{500}{db}$ for d-left hash since it needs to fetch d buckets. We count both on-chip and off-chip space requirement per prefix for all three hashing schemes. For the AP-Hash, the on-chip space is for building the HIT, and the off-chip space includes both the routing table and the HIT mapping table (assuming 64bits per prefix).

We can make several observations from Table III. First, the AP-Hash achieves the highest routing throughput of 250 millions IP lookups per second. It has over three times higher throughput than the throughput of using the single hash and the d-left hashes. This is because the AP-Hash not only reduces the bucket size to be nearly optimal but also needs to fetch only one bucket for each lookup. Second, although the d-left hashes have small bucket size, it needs to fetch d buckets so that the routing throughput is reduced to $1/d$., Third, even with the HIT Mapping Table, the AP-Hash still requires smaller off-chip memory space than the single-hash and the 2-left hash. The 4-left hash needs the smallest off-chip memory, however, its routing throughput is the worst. Lastly, the AP-Hash needs small on-chip space, 1 bit per prefix to build the HIT.

Next, we simulate the impact of using different number of buckets. The results are given in Figure 3 where the horizontal axis is the number of buckets and the vertical axis is the average throughput. We can observe that the AP-Hash has about three times routing throughput than any other hashing schemes over the entire range of buckets.

In the third experiment, we study how our AP-Hash

TABLE IV
ROUTING TABLE UPDATES UNDER AP-HASH AND AP-HASH-E

	AP-Hash	AP-Hash-E
number of buckets	350,000	350,000
forced to re-setup	221	1.01
bit fetched per add	305.2	305.0
bit updated per add	130.6	129.6

and its enhancement AP-Hash-E handles the routing table updates. We initialed our hash table with the routing table as286 dumped at January 1st, 2010 from [1] and use the update trace during January 2010 to simulate the re-setup frequency. The update trace contains 1.5 million additions and 1.5 million deletions and 8 million updates on the output port with $m = 350,000$. As shown in Table IV, the AP-Hash needs to re-setup the routing table about 221 times during the whole month, which is about 7 times a day. The AP-Hash-E only needs to re-setup once for the entire month. The bits fetched and updated for each prefix addition for AP-Hash and AP-Hash-E are very close. The results show that the enhanced AP-Hash-E rarely need to re-setup the entire routing table.

V. RELATED WORK

There are three main categories of IP lookup approaches which support the longest prefix match (LPM) requirement, including Ternary Content Addressable Memories (TCAM)[2], trie-based searches [3], and hash-based approaches [8], [7], [4], [9], [5]. TCAMs are the special-designed memory modules which support simultaneously searching to all the stored contents. Although the delay for TCAM is low, it requires significant space and power and is not suitable for large routing tables. Trie-based approaches use one-bit tree (trie) to organize the prefixes. They consume fewer power and storage space, but needs multiple consecutive off-chip memory accesses which result in long lookup delays and low routing throughput.

Hashed-based approaches use hash tables to organize the prefixes. It is power-efficient and capable of handling large routing tables. However, there are two fundamental difficulties for hash-based approaches: hash collisions and inefficiency in handling the LPM function. Multiple hashing functions can be used and each prefix can be placed into the bucket with fewest prefixes among the d buckets with the d hash functions [13], [8]. Cuckoo [9] and Peacock [15] further reduce the bucket size by relocating the prefixes from the long buckets to short ones. However, all those approaches need to search multiple buckets.

Many works focus on handling the LPM requirement in the hash-based approaches. Their goals are orthogonal to what we do in this paper. The approaches, such as Controlled Prefix Expansion [6], Bloom filter and its invariance [7], [4], can be combined with our approaches.

An IP-lookup approach based on prefect hash functions by using the counted Bloom filter is proposed in [10]. Although it can reach very high routing throughput, it needs about 8.6 bit per key on chip and it cannot handle the routing table updates. The DM-Hash approach in [16] achieves one bucket access with near optimal bucket size by using a small on-chip index table. However DM-Hash also has difficulties to handle the routing table updates and its on-chip space is larger than our AP-Hash.

VI. CONCLUSION

This paper studies the hash-based IP lookup problem and focus on solving two essential issues. The first one is how to minimize the off-chip routing-table access by reducing both the size of hashing buckets and the number of buckets that needs to be fetched for each routing lookup. We introduce a new hashing scheme, AP-Hash, which can balance the hashing buckets, hence reduce the size of the bucket by providing multiple choices of hash functions in placing each prefix. To avoid accessing multiple buckets during the lookup, a small Hash Index Table (HIT) is implemented using the fast on-chip memory in network routers to record the selected hash function for each prefix. The second issue is how to handle routing table updates. A HIT mapping table which records the mapping of all the prefixes into the HIT is saved off-chip. Such a table allows partial updates of the HIT and routing table to keep the buckets balanced when new prefixes are inserted. Performance evaluations of real routing tables and update traces show that our approach can maintain a constant routing throughput of 250M lookups per second and only needs to re-setup the whole table once in a month.

REFERENCES

[1] "Routing Information Service," *http://www.ripe.net/ris/*, 2009.

[2] M. Akhbarizadeh, M. Nourani, D. Vijayasarathi, and P. Balsara, "Pcam: A Ternary Cam Optimized for Longest Prefix Matching Tasks," *In Proc. of IEEE ICCD*, 2004.

[3] W. Jiang, Q. Wang, and V. Prasanna, "Beyond TCAMS: An SRAM-based Parallel Multi-Pipeline Architecture for Terabit IP Lookup," *In Proc. of IEEE INFOCOM*, 2008.

[4] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," *In Proc. of ACM SIGCOMM*, 2006.

[5] H. Song, F. Hao, M. Kodialam, and T. Lakshman, "IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," *In Proc. of INFOCOM*, 2009.

[6] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Transactions on Computer Systems*, 1999.

[7] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor., "Longest Prefix Matching using Bloom Filters," *In Proc. of ACM SIGCOMM*, 2003.

[8] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *In Proc. of INFOCOM*, 2001.

[9] S. Demetriades, M. Hanna, S. Cho, and R. Melhem, "An Efficient Hardware-based Multi-hash Scheme for High Speed IP Lookup," *In Proc. of IEEE HOTI*, Aug 2008.

[10] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," *Proceedings of IEEE Symposium on Information Theory*, 2006.

[11] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," *WADS*, 2007.

[12] A. Czumaj and V. Stemann, "Randomized allocation processes," *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, 1997.

[13] Y. Azar, A. Broder, and E. Upfal, "Balanced Allocations," *In Proc. of ACM STOC*, 1994.

[14] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A performance study of hashing functions for hardware applications," *Proceedings of 6th International Conference Computing and Informationy*, 1994.

[15] S. Kumar, J. Turner, and P. Crowley, "Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms," *In Proc. of IEEE INFOCOM*, 2008.

[16] Z. Huang, D. Lin, S. Chen, J. Peir, and I. Alam, "Fast Routing Table Lookup Based on Deterministic Multi-Hashing," *In Proc. of IProceedings of the 18th IEEE ICNP*, 2010.