Fast Routing Table Lookup Based on Deterministic Multi-hashing

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

***Abstract*—New generations of video, voice, high-performance computing and social networking applications have continuously driven the development of novel routing technologies for higher packet forwarding speeds to meet the future Internet demand. One of the fundamental design issues for core routers is fast routing table lookup, which is a key problem at the network layer of the Internet protocol suite. It is difficult to scale the current TCAM-based or trie-based solutions for future routing tables due to increasing table size, longer prefix length, and demands for higher throughput. This paper focuses on hash-based lookup solutions that have the potential of offering high throughput at one memory access per packet. We design the first deterministic multi-hashing scheme with small indexing overhead, which evenly distributes address prefixes to hash buckets for routing-information storage. We minimize both the size of each bucket and the number of buckets that need to be fetched to the network processor for packet forwarding. Consequently, near-optimal routing throughput is achieved. Performance evaluations demonstrate that the proposed deterministic multi-hashing scheme can maintain a constant lookup rate of over 250 million packets per second with today's commodity SRAM, which is much faster than the existing hashing schemes.**

I. Introduction

Routing-table lookup is one of the key functions at the network layer of the Internet protocol suite. As multimedia content, IPTV, cloud computing, distributed data center networks, and peer-to-peer video streaming applications are increasingly deployed on the Internet, the line speed of the Internet core routers will soon exceed 100 Gbps or even terabits per second in the near future [1]. A router is expected to be capable of forwarding more than 150 million packets per second on a single network connection [2]. To sustain such a high speed, special network processors are designed to implement most routing operations in hardware. One basic function is to look up in a routing table for an entry that matches the destination IP address of an arrival packet. Each table entry contains an address prefix and the associated routing information for the output port and the next hop IP address. An address prefix (or prefix in short) is 8 to 32 bits long for IPv4 and 16 to 64 bits long for IPv6. Because of variable prefix lengths, the router performs longest-prefix match to obtain the routing information.

The number of address prefixes in today's Internet routing tables exceeds 300K and is growing continuously [3]. When the Internet moves to IPv6 in the future, the size of the routing tables will expand tremendously. Such a large table cannot fit in the limited on-chip memory of a network processor. Hence, the routing table is expected to be stored on off-chip SRAM, and its entries are fetched in blocks from the SRAM to the network processor chip on demand. Off-chip memory access is the bottleneck in this routing architecture. We must minimize the number of memory accesses that is needed to forward a packet. Moreover, it is highly desirable that the time it takes to route a packet is a constant. If the worst-case lookup time is much longer than the average, then the network processor has to maintain a large on-chip packet queue, which increases not only the overhead for the chip's hardware resources but also the burstiness in the communication traffic. More importantly, it results in packet drops if the queue is overflowed. These problems are particularly undesirable for *real-time* voice/video delivery because retransmission of lost data does not help.

The current TCAM-based and trie-based approaches for routing-table lookup face serious challenges in satisfying the future demands. Due to power and cost reasons, TCAMs are only suitable for small routing tables [4], [5], [6]. The trie-based search algorithms [7], [8], [9] require many memory accesses in the worst case and its routing time is variable, depending on the length of the matching prefix. On-chip caching has been considered for fast lookups of recently used prefixes [10]. It does not reduce the worst-case routing time of the trie-based algorithms. Moreover, recent study shows that Internet traffic does not have strong locality in its packet stream [2]. The reason is that packets from millions of different sources to millions of different destinations may be mixed together when they arrives at an Internet core router. Consecutive packets from a source to a destination may be interleaved by thousands of packets from other sources to other destinations. This

978-1-4244-8645-8/10/$26.00 ©2010 IEEE

results in poor locality and severely limits the effectiveness of on-chip caching.

This paper investigates the hash-based approach that maps the prefixes in a routing table into an array of buckets in the off-chip SRAM. Each bucket stores a small number of prefixes and their routing information. The network processor hardware can be efficiently designed to retrieve one bucket in each memory access and search the bucket for the routing information associated with the matching prefix. There are two challenging problems that must be solved. First, the prefixes in the routing table have variable lengths. For a specific destination address, we do not know how many bits we should extract and use as the prefix to identify the bucket for retrieval. Second, the bucket size is determined by the largest number of prefixes that are mapped to a bucket. If one hash function is used to map prefixes to buckets, it will result in a large bucket size due to the uneven distribution of prefixes in buckets. For a fixed bandwidth between the SRAM and the network processor, a larger bucket size means a long access time for bringing a bucket to the network processor.

Tremendous progress has been made to address the first problem. Using pipeline and multi-port multi-bank on-die memory, a series of novel solutions based on Bloom filters [11], prefix expansion [12] or distributed Bloom filters [2] are able to determine the prefix length for hash-based routing table lookup in a few or even one clock cycle. This leaves the fetch of routing information (in buckets) from the off-chip SRAM to the network processor as the throughput bottleneck of the lookup function. Hence, the focus of this paper will be the second problem. Given an address prefix, we want to quickly identify the bucket or buckets that may contain its routing information and fetch them to the processor. In order to maximize the lookup throughput, the key is to reduce the number of buckets and the size of each bucket that needs to be brought to the processor for each packet.

The existing multi-hashing schemes map each prefix in the routing table to multiple buckets and store the prefix only in the least-loaded bucket [13], [14], [15], which balances the storage load among the buckets and results in a smaller bucket size. However, in order to find the routing information of a given prefix, the network processor has to fetch multiple buckets because it does not know exactly which bucket a prefix is stored. Ideally, we want to minimize the bucket size and at the same time fetch only one bucket for every packet, such that the optimal routing throughput can be achieved.

In this paper, we propose a *deterministic multi-hashing scheme* to maximize the lookup throughput. The new scheme uses an index table to encode information that allows the network processor to determine exactly which bucket holds a given prefix. The size of the index table is made small, making it easy to fit in on-die cache memory. Another unique advantage of having the index table is that the values stored in the table can be chosen to balance the prefix distribution among buckets and consequently minimize the bucket size, which in turn reduces the space overhead for storing the routing (or forwarding) table. Exactly one bucket is retrieved when the network processor forwards a packet. Hence, the routing-table lookup time is a constant. Because of the small bucket size, the lookup throughput is near optimal. We believe this is the first multi-hashing scheme with small indexing overhead, which deterministically places prefixes in buckets and yet has the flexibility of balancing the load across all buckets. While our focus is about routing, the proposed multi-hashing scheme is very general and can be applied to other applications involving large table lookups.

Our performance evaluation is based on five recent routing tables on Internet backbone routers [3]. The experimental results demonstrate that the proposed multi-hashing scheme is capable of delivering a constant lookup rate of over 250 millions per second. The new scheme significantly outperforms the existing multi-hashing schemes.

The rest of the paper is organized as follows. Section II describes the background of our research. Section III proposes a new multi-hashing scheme for fast routing-table lookup. Sections IV and IV-E give evaluation methodology and experimental results, respectively. Section V discusses the related work. Finally, Section VI draws the conclusion.

II. Hash-based Routing Schemes

In this section, we describe the problem and challenge of hash-based table lookup and point out the limitation of the existing work.

A. Problem and Challenge

A high-speed router needs two algorithms to perform efficient packet forwarding. The *setup algorithm* organizes the routing table information in an indexing data structure to enable fast lookup. The *lookup algorithm* searches the indexing structure for the address prefix that matches a given destination address. Each prefix in the indexing structure is stored together with the routing information, including the output port and the IP address of the next hop router.

In this paper, we consider the hash-based indexing structure. Suppose we have n address prefixes that are hashed to m buckets, each occupying a fixed size of memory that can store a certain number of address prefixes and the corresponding routing information. When the router receives a packet, it first determines the prefix length for lookup [11], [2], extracts a prefix of that length from the destination address, and then hashes the prefix to find the relevant bucket. The bucket is retrieved to the network processor chip (NPC) from an off-chip SRAM. The NPC searches the bucket for the prefix and the associated routing information, based on which the packet is forwarded. Some

TABLE I
NOTATIONS

Symbol	Meaning
n	Number of prefixes
m	Number of buckets, $m \leq n$
B	Bandwidth in terms of number of bits per second from the off-chip SRAM to the NPC
s	Bucket size in terms of number of bits
t	Number of buckets to be fetched for each table lookup
k	Number of hash functions used in a multi-hashing scheme
l	Number of bits for storing store a prefix and its corresponding routing information
Ω	Bucket size in terms of number of prefixes that can be stored, i.e., $s = \Omega \times l$

TABLE II
ROUTING THROUGHPUT COMPARISON

	Routing Throughput
General formula	$\frac{B}{ts}$
Optimal	$\frac{mB}{nl}$
NM-Hash (when k is large)	$\sim \frac{mB}{knl}$
DM-Hash (when k is large)	$\sim \frac{mB}{nl}$

hashing schemes require fetching multiple buckets to route a packet [13], [16].

The operation that the NPC performs to search a bucket for a given prefix can be made very efficient through hardware pipelining. The bottleneck that limits the routing throughput is the communication from the SRAM to the NPC. Because the NPC continuously processes the arrival packets, its hashing and searching operations for some packets overlap with the fetch of SRAM data for other packets. Due to the parallel nature among different stages of packet processing, the routing throughput is entirely determined by the bottleneck operation. Let B be the maximum bandwidth at which the NPC can fetch data from the SRAM and s be the bucket size. If the NPC needs to fetch t buckets in order to forward a packet, then the routing throughput is bounded by $\frac{B}{ts}$ packets per second. Notations are given in Table I for quick reference.

We should minimize ts in order to maximize the routing throughput. The minimum number of buckets that the lookup algorithm needs to fetch is one. Let l be the number of bits it takes to store a prefix and its associated routing information. The minimum bucket size for storing all n prefixes in the m buckets is $\frac{nl}{m}$, in which case the prefixes are evenly distributed among the buckets and the space in each bucket is fully utilized. When both the number of fetched buckets and the size of each bucket are minimized, the routing throughput is maximized at $\frac{B}{1 \times \frac{nl}{m}} = \frac{mB}{nl}$. However, as we will discuss next, minimizing both of the above quantities is a difficult task that has not been accomplished in the prior literature.

B. Non-deterministic Multi-hashing Schemes

It is well known that a single hash function cannot evenly distribute prefixes in the buckets [13]. We perform a simulation with n = 1,000,000 and m = 300,000. The prefixes are randomly generated. We assign IDs from 0 to $m-1$ to the buckets. Each prefix p is stored at a bucket whose ID is $H(p)$, where $H(\cdots)$ is a hash function with a range of [0, $m-1$). The distribution of the prefixes among the buckets is presented in the left plot of Figure 1. The plot shows that some buckets hold much more prefixes than others. The largest number of prefixes that a bucket has to hold (denoted as Ω) is 15, which is 4.5 times the average number, $\frac{n}{m}$. Unfortunately, the bucket size s is determined by Ω. It has to be as large as Ωl in order to avoid overflowing. Because the routing throughput $\frac{B}{ks}$ is inversely proportional to the bucket size and thus inversely proportional to the value of Ω, the actual throughput achieved under a single hash function will be less than one fourth of the optimal value $\frac{mB}{nl}$. Moreover, the total memory consumption is sm, which increases linearly in the value of s (or the value of Ω).

To solve the uneven distribution problem, the main method adopted in the prior research is to use multiple hash functions. Let $H_1(p), \cdots, H_k(p)$ be k different hash functions. The setup algorithm decides the placement of the prefixes sequentially. When processing a prefix p, it uses the hash functions to map the prefix to k buckets whose IDs are $H_1(p), \cdots, H_k(p)$. It then stores p in the bucket that currently has the fewest number of prefixes. As shown in the right plot of Figure 1, when we use more hash functions by increasing the value of k, the prefixes are more evenly distributed among the buckets and the value of Ω decreases, which means a smaller bucket size.

Using the above idea, various sophisticated multi-hashing schemes were proposed to minimize the bucket size [17], [18], [13], [14], [16]. They are called *non-deterministic muti-hashing (NM-hash) schemes* because we do not know exactly which bucket a prefix is stored even though we know that it must be one of the k buckets that the prefix is mapped to. Hence, the lookup algorithm has to fetch all k buckets and the NPC searches them in parallel. That is, $t = k$ for any NM-hash scheme. Even at the optimal bucket size $\frac{nl}{m}$, the routing throughput is only $\frac{mB}{knl}$, which is one kth of the optimal, where k is an integer greater than 1. The dilemma is that in order to reduce s (the bucket size) for better routing throughput, we have to increase k (more hashing functions), which however will reduce the routing throughput. Peacock [15] accesses a variable number of buckets per packet, from 1 to k, resulting in a variable lookup time.

The goal of this paper is to keep the benefit of multi-hashing, while only requiring the lookup algorithm to fetch one bucket. To achieve this goal, we have to abandon the

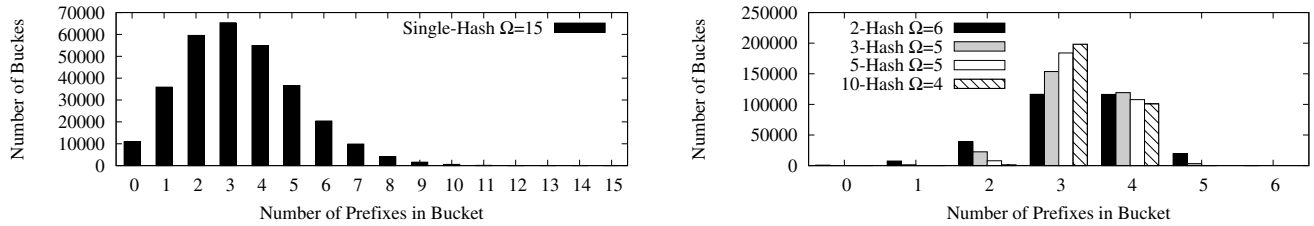

Fig. 1. *Left plot*: the distribution of 1,000,000 prefixes in 300,000 buckets when a single hash function is used. *Right plot*: the distribution of 1,000,000 prefixes in 300,000 buckets when multiple hash functions are used. The distribution is more even if most buckets have similar number of prefixes around the average (which is 3.33). In this example, even distribution means that most buckets should have 3 or 4 prefixes and the minimum value of Ω is 4.

TABLE III
NOTATIONS USED IN DM-HASH

Symbol	Meaning
x	Number of entries in the index table
e_i	An entry in the index table
G_i	A group of prefixes
N_i	Hash neighborhood of G_i
E_i	Given a progressive order of the entries, $e_1, e_2, \cdots, e_x$, we define $E_i = \{e_1, \cdots, e_i\}$.

idea behind the non-deterministic multi-hashing schemes, and invent a new method for deterministic multi-hashing.

III. A NOVEL DETERMINISTIC MULTI-HASHING SCHEME (DM-HASH)

In this section, we propose a deterministic multi-hashing scheme to distribute the prefixes in the buckets for near optimal routing throughput.

A. Deterministic Multi-hashing

In our *deterministic multi-hashing (DM-hash) scheme*, the setup algorithm uses multiple hash functions to decide where to place a prefix (which may end up in any of the m buckets), yet the lookup algorithm has the information to determine the exact bucket where the prefix is stored. Below we give a design overview of our DM-hash scheme.

Because a prefix may be stored in any of the m buckets, we must store some information for the lookup algorithm to know which bucket it actually resides. The simplest solution is to build an *index table* that has an entry for each prefix, keeping the ID of the bucket where the prefix is stored. However, this naive approach does not work because the index table must be stored on the NPC where the lookup algorithm is executed, but it has the same number n of entries as the routing table itself, which makes it too big to fit on the NPC. The counting Bloom filter used in [19] also have a large number of entries, equal to the number m of hash buckets. To fit in the on-die cache memory, we want to reduce the size of the index table, which means that each table entry has to encode the mapping information for multiple prefixes.

DM-hash is designed to work with any size of the index table, which contrasts with Chisel's requirement of more than n entries [20] due to a fundamentally different design (see Section VI for more details). Let $x(\ll m)$ be the number of entries in the index table. In DM-hash, each entry is an integer of $\log_2 m$ bits long, where m (the number of buckets) is chosen to be a power of 2. Instead of mapping each prefix directly to the buckets as the previous schemes do, we map each prefix to k entries of the index table. The XOR of the k entries gives the ID of the bucket in which the prefix will be stored; in order to improve the mapping randomness, we may also use the hash of the XOR result for the bucket ID. Because the index table is small and located on chip, its access is much faster than the bottleneck of off-chip memory access. Each different way of setting the values for the x entries in the index table results in a different placement of the prefixes in the buckets. There are m^x different ways, which represent a large space in which we can search for a good setup of the index table for balanced prefix distribution. The key challenge is how to find a good setup of the index table without exhaustively searching all m^x possible ways. Our solution is described below.

The setup algorithm of the proposed DM-hash scheme divides the n prefixes intro x disjoint groups, denoted as $G_1, G_2, \cdots, G_x$. Each prefix in G_i is mapped to k entries in the index table, which is called the *hash neighborhood of the prefix*. The union of the hash neighborhoods of all prefixes in G_i forms the *hash neighborhood of the group*, denoted as N_i, $1 \leq i \leq x$. We sort the entries of the index table in a so-called progressive order, denoted as $e_1, e_2, \cdots, e_x$, which has the following properties: Consider an arbitrary $i \in [1, x]$. The first property is that e_i must be in the hash neighborhood of every prefix in G_i. It implies that the value of e_i affects the placement of every prefix in G_i. Let $E_i = \{e_1, \cdots, e_i\}$. The second property is that E_i contains the hash neighborhood of G_i, namely, $N_i \subseteq E_i$. After the values of entries in E_i are assigned, the placement of all prefixes in G_i to the buckets must have been determined. This is obvious because according to the second property, a prefix in G_i are only mapped to k entries in E_i. Since E_1, ..., E_{i-1} are subsets of E_i, after the values of entries in E_i are assigned, the placement of all prefixes in G_1, ..., G_{i-1} must also be determined. We will explain

how to divide prefixes into groups and how to establish a progressive order among the entries in the index table in Section III-B.

Unlike the NM-hash schemes [17], [18], [13], [14], [16] that map one prefix to the buckets at a time, our setup algorithm maps a group of prefixes to the buckets at a time. Following the progressive order, the algorithm sequentially assigns values to the entries of the index table. Suppose it has assigned values to entries in E_{i-1} and the next entry to be assigned is e_i. Based on the properties of the progressive order, we have known that the placement of prefixes in groups $G_1, \cdots, G_{i-1}$ has already been determined. The ID of the bucket in which a prefix in G_i will be placed is the XOR of e_i with $(k-1)$ other entries in E_{i-1}. Therefore, once we assign a value to e_i, the placement of all prefixes in G_i will be determined. The entry e_i may take m possible values (from 0 to $m-1$). Each value results in a different placement of the prefixes in G_i to the buckets. The setup algorithm will choose the value that achieves the best balanced distribution of prefixes in the buckets (see Section III-C for details). The algorithm repeats the process until the values of all entries are chosen.

In the DM-hash scheme, the lookup algorithm hashes the prefix extracted from a destination address to k entries in the index table, and then it XORs the entries to identify a bucket, which will be fetched from the SRAM to the NPC. The setup algorithm maps a group of prefixes to the buckets at a time. It tries m different ways to find the best mapping for each group. In the NM-hash schemes, the lookup algorithm hashes the prefix directly to k buckets, which will be fetched. Their setup algorithms map one prefix to the buckets at a time. They try k different ways to find the best mapping for the prefix. Both DM-hash and NM-hash schemes do well in balancing the numbers of prefixes stored in the buckets. Their Ω values can be made close to the optimal, $\frac{n}{m}$. Their memory overhead for storing the routing information in the buckets can thus be made close to the optimal, $m \times \frac{n}{m} l = nl$, as well. However, the DM-hash scheme only requires fetching one bucket for each table lookup. Hence, its routing throughput is also close to the optimal value of $\frac{mB}{nl}$, which compares favorably with $\frac{mB}{knl}$ of the NM-hash schemes. The additional overhead for the DM-hash scheme is a small on-die index table of size $x \log_2 m$, where x can be made much smaller than m (see Section III-D).

B. Progressive Order

DM-hash maps each prefix to k entries in the index table. The set of prefixes that are mapped to an entry is called the *associated group* of the entry. Clearly, an entry is in the hash neighborhood of any prefix in its associated group. If we look at the associated groups of all entries as a whole, each prefix appears k times in them. The problem is to remove prefixes from groups such that (1) each prefix appears exactly once in the groups and (2) when we put the entries in a certain order, their associated groups will satisfy the properties for progressive order.

We present a simple algorithm to solve the above problem: Randomly pick an entry as the last one e_x in the progressive order to be constructed. Remove the prefixes in its current associated group G_x from all other groups. Then, randomly pick another entry as the second last one e_{x-1} and remove the prefixes in its associated group G_{x-1} from all other groups. Repeat the same operation to randomly select entries $e_{x-2}, \cdots, e_1$, and finalize the prefixes left in $G_{x-2}, \cdots, G_1$. The resulting sequence, $e_1, \cdots, e_x$, together with their associated groups, represents a progressive order. This can be proved as follows: Consider an arbitrary entry e_i and an arbitrary prefix p in G_i. Based on the initial construction of G_i, we know that e_i must be in the hash neighborhood of p. We prove the hash neighborhood of p is a subset of E_i by contradiction. Suppose the neighborhood of p contains e_j, where $j > i$. Hence, p is initially in G_j. When e_j is selected, p is removed from the associated groups of all other entries and therefore p must not be in G_i, which leads to the contradiction. Because the hash neighborhood of p is a subset of E_i for any prefix p in G_i, it must be true that $N_i \in E_i$. This completes the proof.

We can improve the above algorithm to balance the group sizes. During the execution of the algorithm, as we remove prefixes from groups, the group sizes will only decrease. Hence, whenever the algorithm selects an entry as e_i, instead of randomly picking one that has not been selected before, it should choose the entry whose associated group has the fewest number of prefixes. Otherwise, if this entry is not chosen for e_i but for e_j later where $j < i$, then the size of its associated group (which is already the smallest) may decrease further.

C. Value Assignment for the Index Table

The setup algorithm sequentially assigns values to the entries in the index table in the progressive order. When it tries to determine the value of an entry e_i, the algorithm iterates through all possible values from 0 to $m-1$. Each possible value corresponds to a different way of placing the prefixes in G_i to the buckets. We define the *bucket-load vector* as the numbers of prefixes in the m buckets sorted in the descending order. Each possible value of e_i results in a bucket-load vector (after the prefixes in G_i are placed in the buckets based on that value). The m different values for e_i result in m bucket-load vectors. The setup algorithm will choose a value for e_i that results in the *min-max vector*, which is defined as follows: Its first number (i.e., the largest number of prefixes in any bucket) is the smallest among all m vectors. If there is a tie, its second number (i.e., the second largest number of prefixes in any bucket) is the smallest among the vectors tied in the first number $\cdots$ If there is a tie for the first j numbers, its (j+1)th number is the smallest among the vectors tied in

the previous numbers. The min-max vector not only has the smallest value of Ω but also has the smallest difference between the largest number of prefixes in any bucket and the smallest number of prefixes in any bucket.

There are x entries. For each entry, the setup algorithm tries m values. For each value, a bucket-load vector is sorted in time $O(m \log m)$. Finding the min-max vector among m bucket-load vector takes $O(m^2)$ time. Each of the n prefixes appears in exactly one group; for each of the m possible values in the entry that the group is associated with, the setup algorithm exams the placement of the prefix, which results in a total time complexity of $O(nm)$. Hence, the time complexity of the setup algorithm is $O(x \times (m \times m \log m + m^2) + nm) = O(xm^2 \log m + nm)$.

D. Analysis

How large should the index table be? We answer this question analytically and support the result with simulation. Let δ_i (δ_{i+1}) be the average number of prefixes in a bucket and Ω_i (Ω_{i+1}) be the large number of prefixes in any bucket after the setup algorithm determines the value of e_i (e_{i+1}) and the prefixes in G_i (G_{i+1}) are placed in the buckets. Let g be the number of prefixes in G_{i+1}. Given the values of δ_i, Ω_i and g as input, we treat Ω_{i+1}, which is the outcome of the setup algorithm once it determines the value of e_{i+1}, as a random variable. Our objective is that whenever $\Omega_i - \delta_i \geq 1$, the subsequent operation of the setup algorithm will make sure that $E(\Omega_{i+1} - \delta_{i+1}) < (\Omega_i - \delta_i)$. Hence, the expected largest number of prefixes in any bucket is controlled within one more than the average number.

Let q be the percentage of buckets that hold no more than δ_i prefixes. We know that the m possible values for e_{i+1} result in m different ways of placing the g prefixes in G_{i+1} to the buckets. To make the analysis feasible, we treat them as m independent random placements of the prefixes. As we will see, such approximation produces results that match what we observe in the simulation. The probability for one placement to put all g prefixes only in buckets currently having no more than δ_i prefixes is q^g. The probability for any one of the m placements to do so is $1 - (1 - q^g)^m$. When this happens, $\Omega_{i+1} = \Omega_i$; otherwise, $\Omega_{i+1} \leq \Omega_i + 1$. We expect that m is chosen large enough such that $g \ll m$ and thus the probability for two prefixes in G_{i+1} to be placed in the same bucket is negligibly small. Hence,

$$\begin{aligned} E(\Omega_{i+1}) &\leq (1 - (1 - q^g)^m)\Omega_i + (1 - q^g)^m(\Omega_i + 1) \\ &= \Omega_i + (1 - q^g)^m \end{aligned}$$

Because $\delta_{i+1} = \delta_i + \frac{g}{m}$, we have

$$E(\Omega_{i+1} - \delta_{i+1}) \leq \Omega_i - \delta_i + (1 - q^g)^m - \frac{g}{m}$$

Hence, our objective is to satisfy the inequality, $(1 - q^g)^m - \frac{g}{m} < 0$. Approximately, we let $q = 1/2$ and $g = \frac{n}{x}$, which means that about half buckets are loaded with an average number of prefixes or below, and that G_{i+1} has about the average number of prefixes among all groups. The inequality becomes

$$m(\frac{1}{2})^{\frac{n}{x}} > -\ln \frac{n}{xm}.$$

When n = 1,000,000 and m = 300,000, the minimum value of x that satisfies the above inequality is 67,180. It agrees with our simulation result in Figure 2. When we choose $x = 60,000$, the simulation shows that Ω = 5. The value of Ω is the closest integer that is larger than one plus the average, $\frac{n}{m}$.

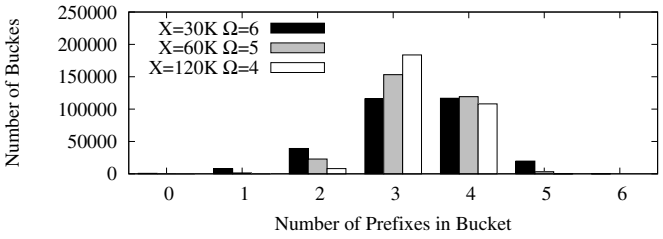

Fig. 2. The distribution of 1,000,000 prefixes in 300,000 buckets under DM-hash with k = 3.

IV. Performance Evaluation

A. Routing Tables

We use five routing tables from the Internet backbone routers: as286 (KPN Internet Backbone), as513 (CERN, European Organization for Nuclear Research), as1103 (SURFnet, The Netherlands), as4608 (Asia Pacific Network Information Center, Pty. Ltd.), and as4777 (Asia Pacific Network Information Center), which are downloaded from [3]. These tables are dumped from the routers at 7:59am July 1st, 2009. They are optimized to remove the redundant prefixes. The prefix distributions of the routing tables are given in Table IV, where the first column is the prefix length and other columns are the number of prefixes that have a given length.

B. Determining the Prefix length

As we have discussed in the introduction, we can use Bloom filters [11] or distributed Bloom filters [2] to determine the length of the prefix that we should extract from a destination address for routing-table lookup. However, the Bloom filters can cause false positives that result in prolonged lookup times for some packets. In our evaluation, we design a simpler approach to determine the prefix length that ensures constant lookup time. We observe in Table IV that the number of prefixes whose lengths are in the ranges of $[8, 18]$ and $[25, 32]$ is very small. They only account for less than 10% of a routing table. We store these prefixes in a TCAM. For the remaining prefixes whose lengths are in the range of $[19, 24]$, we perform prefix expansion [12] to reduce the number of different lengths. When we say "expansion to i" where $i \in [19, 24]$, we mean that

TABLE IV
PREFIX DISTRIBUTIONS

Length	as286	as513	as1103	as4608	as4777
8	15	15	15	15	15
9	10	10	10	10	10
10	18	18	18	18	18
11	46	46	46	46	46
12	136	135	134	136	135
13	300	302	301	301	304
14	500	504	502	501	501
15	1013	1014	1017	1009	1012
16	9514	9507	9540	9486	9521
17	4245	4300	4257	4253	4259
18	7298	7293	7320	7260	7299
19	15725	15988	15706	15947	15908
20	19295	19440	19400	19283	19353
21	19101	19214	19197	19200	19206
22	24925	25094	25044	25068	25056
23	24500	24650	24592	24729	24626
24	150125	151799	151276	153055	151289
25	6	531	146	741	898
26	10	542	212	853	1036
27	3	506	148	460	560
28	17	423	147	343	147
29	14	800	86	642	8
30	45	554	115	325	7
31	0	0	0	4	0
32	28	8318	35	0	8
Sum	276889	291003	279264	283685	281222

each prefix whose length is shorter than i is replaced by multiple prefixes of length i, whose combined address space is equivalent to what the original prefix represents. Prefix expansion reduces the number of different lengths but increases the number of prefixes, which is shown in Figure 3. After expansion, the prefixes will be stored in the hash buckets based on whichever hashing scheme under use. To determine the bucket for a prefix (whose length is now in the range of $[i, 24]$), we only use the first i bits.

To look up for a given destination address, the network processor checks the TCAM and the hash-based routing table in parallel. For the hash-based routing table, it uses the first i bits of the destination address to determine a bucket or buckets (in the case of NM-hash schemes) to fetch.

C. Hashing Schemes under Comparison

We compare three hashing schemes in routing-table lookup: *single-hash*, which uses one hash function to map the prefixes to the buckets; *NM-hash*, which uses two hash functions and is designed based on [14]; *DM-hash*, which is proposed in this paper and also uses two hash functions. The reason we use only two hash functions is that more hash functions will significantly increase the computation and throughput overhead[1] but only marginally reduce the bucket size. We use DM-hash(x) to denote DM-hash whose index table has x entries. To make it easy for hardware implementation, we use two simple hash functions: the first One simply takes the low-order bits of a prefix as the hash output. The second one takes the XOR of the low-order bits and the adjacent high-order bits as the hash output. Namely, $H(p) = x_1x_2\cdots x_i \bigoplus y_1y_2\cdots y_i$, where p=$x_1x_2\cdots x_iy_1y_2\cdots y_i\cdots$. The single-hash scheme uses the second function. The number of bits in the output is determined by the number of bits that is required.

[1]Recall that the throughput $\frac{mB}{knl}$ of an NM-hash scheme is inversely proportional to k.

D. Experiment Process

In each experiment, we will select a routing table and a hashing scheme. We vary the number m of hash buckets from 16K to 2M, each time doubling the value of m. We find the prefixes in the routing table whose lengths are between 19 to 24, and expand them to a certain length i. For each value of m (16K, 32K, 64K, 128K, 256K, 512K, 1M, or 2M), we distribute these prefixes to the buckets based on the selected hashing scheme, and determine the bucket size s, which in turn determines the bandwidth it takes to fetch a bucket to the network processor and consequently determines the routing throughput (see the next section for more details). Eight different values of m will give us eight data points.

The bucket size is automatically determined after all prefixes are placed in the buckets. The total off-chip SRAM overhead can be characterized by the maximum number $\frac{sm}{l}$ of prefixes that all buckets can together hold, where sm is the memory (in terms of number of bits) occupied by all buckets, and l is the number of bits to store one prefix and its associated routing information. We define the following metric:

$$\textit{Memory/Prefix Ratio} = \frac{\frac{sm}{l}}{n} = \frac{sm}{ln},$$

which is the ratio of the maximum number of prefixes that can be stored to the number of prefixes that are stored. It will be used when we present the experimental results. Clearly, when the number n of prefixes is given, the memory/prefix ratio characterizes the space overhead.

E. Performance Results

1) Routing Throughput: To calculate the routing throughput, we let each <prefix, output port> pair occupies 5 bytes (24 bits for the prefix and 16 bits for the output port). The current QDR-III SRAMs runs at 500MHz and supports 72-bit read/write operations per cycle [21]. Let Ω be the largest number of prefixes that a hashing scheme maps to a bucket. The routing throughput (or lookup rate) can be computed as:

$$Throughput = \frac{500}{\lceil \frac{40\times\Omega}{72} \rceil} M/Sec,$$

where $40 \times \Omega$ is the bucket size.

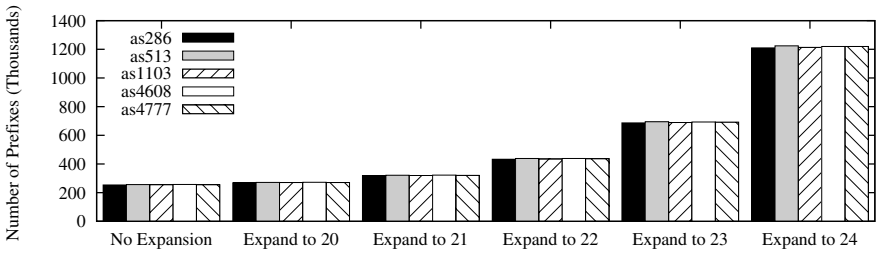

Fig. 3. Number of prefixes after expansion.

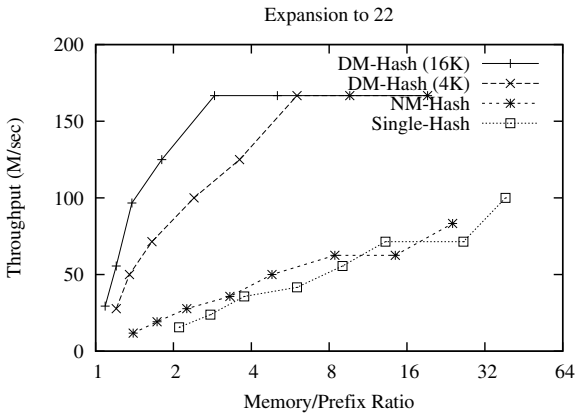

Fig. 4. Routing throughput comparison when the prefixes are expanded to 22 bits.

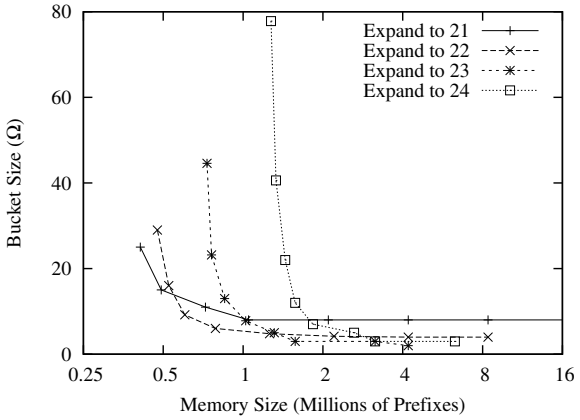

Fig. 6. Bucket size under DM-hash with respect to memory size.

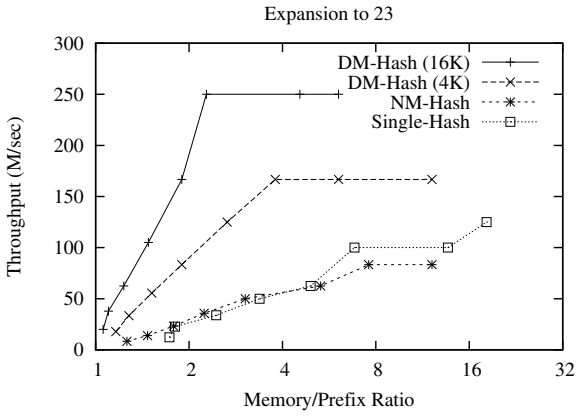

Fig. 5. Routing throughput comparison when the prefixes are expanded to 23 bits.

Our evaluation results for routing throughput are shown in Figure 4, where the prefixes in the routing table are expanded to 22 bits. The horizontal axis is the memory/prefix ratio, and the vertical axis is the number of lookups per second. Each hashing scheme has eight data points, which are the average among five routing tables. As predicted, the single-hash scheme has the lowest throughput. The NM-hash scheme is only marginally better than the single-hash scheme. The reason is that, although the NM-hash scheme has a smaller bucket size, it has to fetch two instead of one bucket, which essentially gives up much of the throughput gain due to smaller buckets. The throughput of the DM-hash scheme is much larger because not only does it reduce the bucket size, but also it fetches only one bucket. DM-hash(16K) has a higher throughput than DM-hash(4K), which indicates that a larger index table helps.

Figure 5 shows similar results under expansion to 23. Surprisingly, the NM-hash scheme performs slightly worse than the single-hash scheme in some cases. The advocates of the NM-hash scheme argue that the buckets can be stored in different memory banks and therefore the operation of fetching multiple buckets can be performed in parallel in one off-chip access time. However, the parallelism will also help the single-hash or DM-hash scheme, which can potentially process the lookups of multiple packets in one off-chip access time. Moreover, it is costly to implement parallelism in off-chip memory access.

2) Bucket Size: Next, we investigate the bucket size that DM-hash has under different levels of memory availability. The evaluation results are shown in Figure 6, where the horizontal axis is the memory size (Ωm) and the vertical axis is the bucket size (Ω). For all prefix expansions from 21 to 24, we observe that there is a throughput-space tradeoff. When the memory size increases, the bucket size decreases, indicating a higher routing throughput.

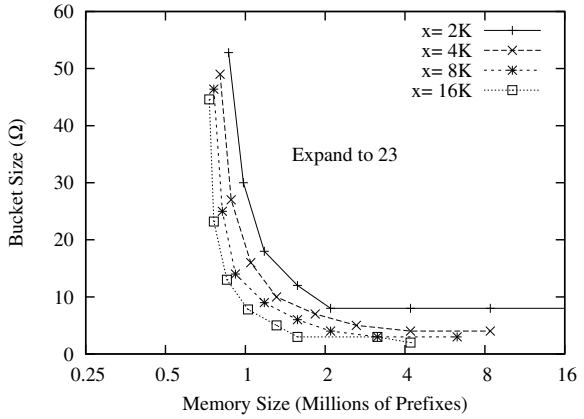

Fig. 7. Bucket size under DM-hash with respect to index-table size.

TABLE V
INDEX TABLE SIZE

	x=2K	x=4K	x=8K	x=16K
number of entries	2048	4096	8192	16384
size (Kilo Bytes)	6KB	12KB	24KB	48KB

3) Size of the Index Table: Finally, we study the performance of DM-hash under different index-table sizes (ranging from 2K to 16K) in Figure 7. The sizes of the index tables are summarized in Table V, assuming that each entry in the index table occupies 24 bits, which supports a maximum number of 2^{24} buckets. As predicted, a larger index table helps to balance the load in the hash buckets better, which means a smaller bucket size. A smaller bucket size helps improve the routing throughput.

V. RELATED WORK

There are three categories of routing-table lookup approaches for longest prefix match (LPM), including Ternary Content Addressable Memories (TCAM) [4], [5], [6], trie-based searches [7], [8], [9], and hash-based approaches [13], [22], [19], [20], [16], [15]. TCAMs are custom devices that can search all the prefixes stored in memory simultaneously. They incur low delays, but require expensive tables and comparators and generate high power dissipation. Trie-based approaches use a tree-like structure to store the prefixes with the matched output ports. They consume a small amount of power and less storage space, but incur long lookup latencies involving multiple memory operations that make it difficult to handle new IPv6 routing tables.

Hashed-based approaches store and retrieve prefixes in hash tables. It is power-efficient and capable of handling a large number of prefixes. However, the hash-based approach encounters two fundamental issues: hash collisions and inefficiency in handling the LPM function. Sophistic hashing functions [23] can reduce the collision using expensive hardware with long delays. Multiple hashing functions allocate prefixes into the smallest hashed bucket for balancing the prefixes among all hash buckets [17], [18], [13]. Cuckoo [16] and Peacock [15] further improve the balance with relocation of prefixes from long buckets. The downside of having multiple choices is that the search must cover multiple buckets. Extended Bloomier Filter [19] places each prefix into multiple buckets and uses a counter to count the number of prefixes in each bucket. During lookup, only the shortest bucket that a prefix is mapped to will be checked. To reduce the space overhead and the length of each bucket, duplicated prefixes are removed. For efficient searches, proper links with shared counters must be established.

Handling LPM is difficult in hash-based lookups. To reduce multiple searches for variable lengths, Control Prefix Expansion (CPE) [12] and its variances [16], [2] reduce the number of different prefix lengths to a small number. The tradeoff is that the number of prefixes is much increased.

Organizing the routing table in a set-associative memory and using common hash bits to allocate prefixes of different lengths into the same bucket reduces the number of memory operations to perform the LPM function [24], [25], [16]. However, by coalescing buckets with multiple prefix lengths, it creates significant hashing collisions and hence increases the bandwidth requirement. Bloom Filter was considered to filter unnecessary IP lookups of variable prefix lengths [11]. Multiple Bloom filters are established, one for each prefix length to filter the need in accessing the routing table for the respective length. Due to uneven distribution of the prefix lengths, further improvement by redistribution of the hashing functions and Bloom filter tables can achieve balanced and conflict-free Bloom table accesses for multiple lengths of prefixes [2]. In these filtering approaches, however, the false-positive condition may cause unpredictable delays due to multiple additional memory accesses. In addition, even though Bloom filters are relatively compact data structures, they still represent significant overhead to the cache memory of the network processor if the false positive ratio is required to be low.

Chisel [20] adopts the Bloomier filter [26] for its indexing structure. It requires a large indexing table to achieve conflict-free hashing. Because each prefix must be hashed to a singleton entry and each entry in the indexing table can serve as a singleton for only one prefix, the table size must be larger than the number of prefixes in the routing table. The former is typically chosen between 1.5 and two times of the latter. Each indexing entry is at least 20 bits long to encode the IDs of up to 1 million buckets. Each forwarding table entry in SRAM is about 5 bytes long, three for an address prefix and two for an output port. Hence, the size of the indexing table in Chisel is between 50% and 100% of the forwarding table (also referred to as the routing table in this paper), making it too large to fit in the on-chip cache memory. Our DM-hash scheme does not require that each prefix must be mapped to a singleton entry. It thus

eliminates the hash collision problem and can work with an index table of any size.

VI. Conclusion

This paper studies the problem of fast routing-table lookup. Our focus is on the problem of minimizing both the size of each hash bucket and the number of buckets that need to be fetched to the processor for packet forwarding. We introduce a new deterministic multi-hashing scheme, which relies on a small intermediate index table to balance the load on the buckets and also allow us to determine exactly which bucket each prefix is stored. The result is a near-optimal throughput of over 250M table lookups per second with today's commodity SRAM.

Although our study is centered around routing-table lookups, the proposed DM-hash is very general and can be applied in other applications involving information storage and retrieval. Applications such as page-table lookups in operating systems, intrusion detections based on virus signature searches, key words matching in Google search engines are possible candidates that can benefit from using the DM-hash scheme developed in this paper.

References

[1] W. D. Gardner, "Researchers Transmit Optical Data At 16.4 Tbps," *InformationWeek*, February 2008.

[2] 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.

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

[4] H. Miyatake, M. Tanaka, and Y. Mori, "A Design for High-Speed Low-Power Cmos Fully Parallel Content-Addressable Memory Macros," *IEEE Journal of Solid-State Circuits*, 2001.

[5] 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.

[6] H. Noda, K. Inoue, M. Kuroiwa, F. Igaue, K. Yamamoto, H. Mattausch, T. Koide, A. Amo, A. Hachisuka, S. Soeda, I. Hayashi, F. Morishita, K. Dosaka, K. Arimoto, K. Fujishima, K. Anami, and T. Yoshihara, "A Cost-Efficient High-Performance Dynamic TCAM with Pipelined Hierarchical Searching and Shift Redundancy Architecture," *IEEE J. Solid-State Circuits*, 2005.

[7] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *ACM SIGCOMM Computer Communication Review*, 2004.

[8] K. Kim and S. Sahni, "Efficient Construction of Pipelined Multibit-tire Router-tables," *IEEE Trans. Computers*, 2007.

[9] 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.

[10] M. J. Akhbarizadeh and M. Nourani, "Efficient Prefix Cache for Network Processors," *In Proc. of IEEE HPI*, Aug 2004.

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

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

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

[14] 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.

[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] 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.

[17] A. Broder and A. Karlin, "Multilevel Adaptive Hashing," *In Pro. of ACM-SIAM SODA*, 1990.

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

[19] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: an Aid to Network Processing," *In Proc. of ACM SIGCOMM*, 2005.

[20] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture," *In Proc. of IEEE ISCA*, 2006.

[21] M. Pearson, "QDRTM-III: Next Generation SRAM for Networking," *http://www.qdrconsortium.org/presentation/QDR-III-SRAM.pdf*, 2009.

[22] X. Nie, D.Wilson, J. Cornet, G.Damm, and Y. Zhao, "IP Address Lookup Using a Dynamic Hash Function," *In Proc. of IEEE CCECE/CCGEI*, 2005.

[23] H. D. House, "HCR MD5: MD5 crypto core family," 2002.

[24] S. Kaxiras and G. Keramidas, "IPStash: A Power Efficient Memory Architecture for IP lookup," *In Proc. of IEEE MICRO*, 2003.

[25] ——, "IPStash: A Set-associative Memory Approach for Efficient IP-lookup," *In Proc. of IEEE INFOCOM*, 2005.

[26] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," *In Pro. of ACM-SIAM SODA*, 2004.