Distilling Router Data Analysis for Faster and Simpler Dynamic IP Lookup Algorithms

Filippo Geraci, Roberto Grossi

***Abstract*— We consider the problem of fast IP address lookup in the forwarding engines of Internet routers. Many hardware and software solutions available in the literature solve a more general problem on strings, the longest prefix match. These solutions are then specialized on real IPv4/IPv6 addresses to work well on the specific IP lookup problem. We propose to go the other way around. We first analyze over 2400 public snapshots of routing tables collected over five years, discovering what we call the *middle-class effect* of those routes. We then exploit this effect for tailoring a simple solution to the IP lookup scheme, taking advantage of the skewed distribution of Internet addresses in routing tables. Our algorithmic solution is easy to implement in hardware or software as it is tantamount to performing an indirect memory access. Its performance can be bounded tightly in the worst case and has very low memory dependence (e.g., just one memory access to off-chip memory in the hardware implementation). It can quickly handle route announcements and withdrawals on the fly, with a small cost which scales well with the number of routes. Concurrent access is permitted during these updates. Our ideas may be helpful for attaining state-of-art link speed and may contribute to setting up a general framework for designing lookup methods by data analysis.**

***Index Terms*— System design, IP lookup algorithms, data analysis, forwarding engines, routing tables.**

I. INTRODUCTION

The IP lookup problem is a recurrent problem in the literature for packet forwarding in the Internet [1]. Routers have to forward lots of packets from input interfaces to output interfaces (*next hops*) based on packet's destination Internet address, called an *IP address*. Forwarding a packet requires an IP address *lookup* at the routing table[1] to select the next hop corresponding to the packet. As routers have to deal with links whose speed constantly improves, the address lookup is considered one of the major bottlenecks in high performance forwarding engines [1], [2]. Other bottlenecks, such as those involved by fair queueing policy and IP switching technology, are well understood and handled [3].

The IP address lookup problem was just considered a simple table lookup problem at the beginning of Internet. In the early 1990s, people realized that routing information would grow enormously and introduced classless inter-domain routing (CIDR) for reducing space by dividing networks into prefixes [4]. In IPv4 [5] the prefixes are binary strings of variable length using the syntax $X.Y.W.Z/L$ to represent the first L bits of the 4-byte word $X.Y.W.Z$, where $8 \leq L \leq 32$. Prefixes can be up to 128 bits in IPv6 [6] (but then have a different syntax). More realistically, we can assume prefix lengths up to 64 bits in IPv6 global unicast addressing [7], since the first 64 bits are crucial for backbone routing while the last 64 bits are for subnet routing, e.g. MAC addresses.

The use of prefixes increases the complexity of the IP address lookup problem. For each packet, more than one prefix in the routing table can match the packet's IP address. In this case, the adopted rule is to take the *longest matching prefix*. Given prefixes $p_1, p_2, \ldots, p_n$, for any binary string x we want to identify the longest p_i that equals the first bits of x, where $1 \leq i \leq n$. For example, let's consider the prefixes in Table I. Both prefixes 192.168.0.0/17 and 192.168.0.0/18 match the IP address 192.168.128.125; hence, the packet is forwarded to next hop 3. We will only consider situations arising with single hops, since dealing with multihops is very similar. No-route-to-host is the special next hop 0 associated with the empty prefix ϵ.

Looking for the longest matching prefix in tables of high-performance routers is a challenging problem. For networks with a link speed of 10 gigabits per second (OC-192), they need to forward up to 33 million packets per second, assuming that each packet is 40 bytes long. A general solution to the longest prefix matching problem (LPM) is not the best choice, since it also has to deal with extreme situations that do not occur in real routing

Istituto di Informatica e Telematica, CNR – Consiglio Nazionale delle Ricerche, 56100 Pisa, Italy (`filippo.geraci@iit.cnr.it`). Part of this work was supported by the Italian CNR.

Dipartimento di Informatica, Università di Pisa, 56125 Pisa, Italy (`grossi@di.unipi.it`). Work supported in part by the Italian Ministry of Research and Education (MIUR).

[1]We will use the term "routing table" to denote what is more properly called a "forwarding table." An actual routing table contains some additional information.

tables. Thus, the resulting algorithms are more involved than a simple table lookup. The IP lookup problem is more peculiar than LPM, because the prefixes stored in the routing tables are not random strings. In this paper we stress the importance of data analysis on real routing tables *before* designing IP lookup algorithms. (We do not consider real traffic analysis due to the difficulty in obtaining public databases for privacy reasons.)

The results in previous work mentioned in Section VI describe the IP address lookup problem in the general terms of LPM. They first discuss how to solve its general form efficiently; then they present experiments to tune the performance of the proposed solutions when applied to the specific IP address lookup problem on real routing tables. Again, we follow the opposite direction in hopes of gaining more insight into the problem. We begin with the experimental analysis performed on public databases of nearly 2400 snapshots of routing tables collected over five years. We identify some new parameters characterizing the (skewed) distribution of prefixes in routing tables. Based upon our findings, we provide a new and simple solution to the IP address lookup problem that circumvents several difficulties posed by the generality of LPM.

Our starting point is the preliminary result based on full expansion and compression of routing tables by Crescenzi, Dardini and Grossi [8]. (It was later referred to as CDG in [9].) To our knowledge CDG is the first to describe a lookup scheme whose design is fully driven by data analysis. A frequently cited survey [1] published in 2001 shows that CDG is almost an order of magnitude faster than its state-of-the-art competitors at that time (see Table 3 in [1]). Even in the worst case, the frequency of lookups with small response time is impressively high and does not depend on the traffic through the router (see Fig. 22 in [1]).

Unfortunately, CDG has some drawbacks. The survey reports that "Schemes using multibit tries and compression give very fast search times. However compression and the leaf pushing technique used do not allow incremental updates. Rebuilding the whole structure is the only solution." Moreover, some authors [9], [10], [11] pointed out some cases in which the space requirement of CDG is too high, possibly causing its performance to suffer in the worst case.

prefix	hop	prefix	hop
`65.10.10.0/24`	1	`192.168.64.0/18`	2
`192.168.0.0/17`	2	`192.168.0.0/32`	4
`192.168.0.0/18`	3	`192.168.0.0/29`	5

TABLE I

In this paper we present a lookup scheme that exploits the original idea of CDG in a novel and even simpler way. We bring to light further properties that allow us to avoid its drawbacks. The main discovery is what we call the *middle-class effect* in real routing tables: even though the majority of prefixes have lengths ranging from 16 to 24, they tend to follow regular patterns. In other words, we have a good chance to store the mapping from all the 2^{32} IP addresses to the next hops into a compact table, so that lookup and update are able to access the table very quickly using indirection. Some of the basic properties that we distill have been implicitly used in some of the previous work to optimize the performance of the proposed solutions. We go the other way around, and design our method using solid data analysis.

The main contributions of our paper on exploring the data analysis can be summarized as follows. First, we save space significantly over CDG since we have a much more stable space occupancy that scales linearly with the table size (e.g., see Fig. 5). We no longer need the run-length encoding (RLE) adopted in CDG, because we organize suitably the prefixes. Second, we improve lookup time by nearly 30% (e.g., see Fig. 7). Third, we can dynamize the table, performing updates quickly without rebuilding the whole structure as previously required. Concurrent access is also permitted while updating.

We think that these contributions are due to the simplicity of our scheme (see Fig.4), whose efficiency is validated by our data analysis. Not only do we reduce space occupancy and make it linearly scalable with the size of routing tables, we also improve lookup time and obtain a fast and scalable update algorithm for supporting announcements and withdrawals. Our update algorithm is robust since we can efficiently bound the worst case, which is important for unauthenticated announcements [12].

Our solution is algorithmic in nature and can be implemented in hardware or software. Available solutions assume processors that make use of fast static random access memories (SRAMs) or ternary content addressable memories (TCAMs). We can use both technologies in our lookup scheme, and refer the reader to [2] for a recent discussion on their advantages and drawbacks. We also attain high throughput by running our lookup scheme on a standard PC. We believe that performance will greatly improve by integrating our scheme to exploit the aforementioned technologies to obtain an embedded system for forwarding packets.

Space is not the main issue; more space-efficient solutions for lookup tables can be found in the literature, but they either have slower access or are difficult to update. Our space occupancy fits current technology, as it requires 1–2Mb of fast memory. We also assert preliminary performance for IPv6 routing tables. Our findings on data analysis can be exploited with other IP lookup methods to improve their performance. Indeed, some of them make implicit use of the data distribution in routing tables. Clearly, our scheme can also be used to solve the general problem of the longest prefix match. However, we do not claim that its performance is as good as in the case of the specific IP lookup problem.

The paper is organized as follows. We illustrate our approach by taking a glimpse into our data analysis in Section II. We show how to perform lookups in Section III and updates with announcements and withdrawals in Section IV. We describe the construction of our lookup table in Section V. We conclude with a reference to state-of-the-art methods in Section VI.

II. Data Analysis of Routing Tables

In this section, we describe our data analysis on routing tables to highlight a useful property of middle-class prefixes (whose length ranges from 16 to 24). We call it the *middle-class effect*. It allows us to reduce both space occupancy and lookup time while efficiently dynamizing the lookup table. While we do not claim to be the first to have exploited this effect, our study explicitly stresses its importance in designing IP lookup tables. We first describe the large data set that we employed from public databases of routing tables for IPv4 in Section II-A. We illustrate the middle-class effect in Section II-B, showing how to exploit it for a two-layer organization in Section II-C. Based on the latter, we describe an implementation of IP lookup tables in Section II-D. We suggest how to scale it to IPv6 in Section II-E.

A. Databases and experimental platforms

We base our analysis on an extensive data set of more than 2400 snapshots of routing tables available

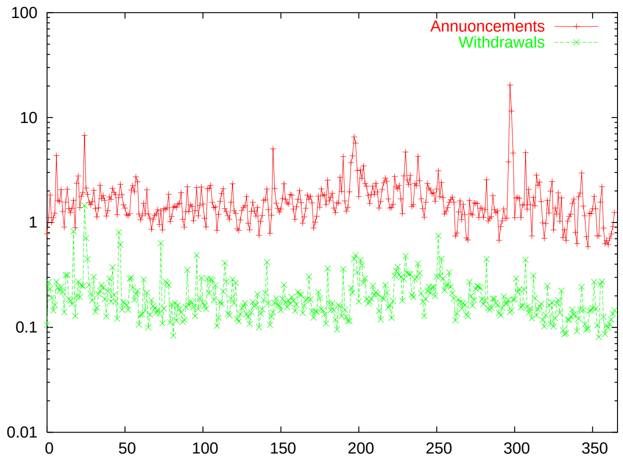

Fig. 1. Millions of daily announcements (top) and of daily withdrawals (bottom) for RIPE NCC, in logarithmic scale on the y-axis. The x-axis reports the 365 days in year 2003.

from public databases, collected over a period ranging from 1998 to 2004. The major source is located at `ftp.merit.edu/ipma/routing_table`, the Internet Performance Measurement and Analysis (IPMA) project (currently dismissed). We also collected all daily data for year 2003, plus some monthly snapshots, from `data.ris.ripe.net`, the Network Coordination Centre of the Réseaux IP Européens (RIPE NCC), router of Amsterdam. We report the figures in Table II.

Some authors singled out individual snapshots that cause the worst-case behavior of CDG in terms of space occupancy; hence, they are good benchmarks for our method as well. Most of these tables have been employed in the experiments [9], [13]. The remaining ones were sent to us [10]. We list them in Table III.

As for the updates, we collected *all* the announcements and withdrawals available for the entire year 2003 on RIPE NCC. In Fig.1, we plot their number in millions (on the y-axis) on a daily basis (on the x-axis). As we can see, the number of withdrawals is an order of magnitude smaller than the number of announcements. On the average, there is approximately one announcement per second; clearly, they arrive in bursts. For example, note

router	#snapshots	from	to
aads	538	10-01-00	05-15-02
mae-east	230	10-01-00	06-01-01
mae-west	618	10-01-99	04-12-02
paix	78	10-01-01	03-10-02
pacbell	576	12-09-98	05-15-02
ripe-ncc	365	01-01-03	12-01-03
ripe-ncc	19	10-10-99	04-01-04

TABLE II

router	date	router	date
aads	05-30-01	oregon-03	07-10-03
att	07-10-03	pacbell	05-30-03
east.attcanada	07-10-03	paix	05-30-01
funet	10-30-97	telstra	03-31-01
mae-west	05-30-01	telus	07-10-03
oregon-01	03-31-01	west.attcanada	07-10-03

TABLE III

the peak of more than 20 million updates on Oct 25–26, 2003. We will use this particular peak for intense benchmarking in Section IV.

As for the lookups, we could not find publicly available traffic traces (for privacy resasons). We instead use random data from previous work [9], as well as synthetic data. We obtain the latter by extending the approach in [14] to generate traffic data according to the distribution of the prefixes of any given routing table T.

To begin with, let S be a stack whose positions are numbered $2, 3, \ldots$, starting from the top. When we push an item into S, the item gets position 2 and the remaining ones are shifted to positions $3, 4$, etc. When we extract an item at position i from S, we shift items in positions $i+1, i+2 \ldots$ so that they occupy positions $i, i+1$, etc.

We generate traffic data using table T, stack S, and a conditional probability $0 < p < 1$ (we set $p = 0.9$ in our experiments). The first IP address is chosen uniformly at random and is pushed into an empty S. We then generate the remaining IP addresses one by one according the following steps:

1) We choose a nonempty item from the stack S, such that the item in position j is picked with probability 2^{-j}, for $j = 2, 3, \ldots$; if we succeed, we output that item. (This happens with probability nearly $1/2$ for a sufficiently large stack.)

2) If no item is chosen in step 1 (again, occurring with probability nearly $1/2$), we toss a biased coin (heads with probability p and tails with probability $1-p$) and consult the following:

2.a—Heads: choose a prefix from T, uniformly at random, pad it with random bits to obtain a length of 32 bits, and output it.

2.b—Tails: output a random IP address uniformly at random.

In all cases, we push the output address onto the top of the stack S, and we extract its copy (if any) from S.

For our experiments we employed two platforms. The first is based on an AMD Athlon XP 1900+ (1.6GHz), 256Mb RAM DDR at 133Mhz, 256Kb L2 cache, 128Kb L1 cache (64 Kb data and 64Kb instructions) running Linux kernel 2.4.22. The second is an Intel Pentium 4 (2Ghz), 512Mb RAM DDR at 133Mhz, 512Kb L2 cache. We plan to extend the experimentation to more platforms (e.g., those based on the PowerPC). We used `gettimeofday` for timings. Since the results are similar, we will report only experimental data for the first platform.

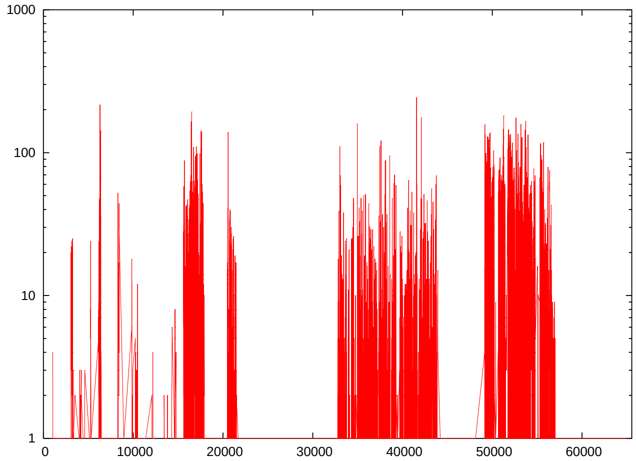

Fig. 2. The number of middle-class prefixes of RIPE NCC is shown on on the y-axis (log scale). The x-axis reports the 2^{16} intervals of the address space, each interval associated with a distinct configuration of the first 16 bits in the addressing. Each vertical bar counts how many prefixes fall within the corresponding interval.

B. Distilling the middle-class effect in routing tables

In order to illustrate our ideas, let's consider any routing table T. We then choose the snapshot of the RIPE NCC router taken on April 1st, 2004, containing 138201 prefixes. Note that analogous properties also hold for the router snapshots in the data set described in Section II-A. What is widely known is the skewed distribution of prefixes from length 1 to 32 in T. Indeed, 98% of the prefix lengths are in the interval $[16 \ldots 24]$, which we call middle-class prefixes. We therefore focus on these prefixes, looking for more insight on their distribution.

We take the address space $[0 \ldots 2^{32} - 1]$ partitioned into equal intervals of size 2^{16}, each interval corresponding to a distinct configuration of the first 16 address bits. For each interval, we count how many middle-class prefixes of T have their first 16 bits corresponding to that interval. Fig.2 shows the resulting frequency of prefixes in these intervals. We obtain a skewed distribution, and this skew is typically a good sign for compressing data (whereas a uniform distribution is bad in this sense).

However, we can get further insight by examining the trie storing all the prefixes in T (see [15] for a definition of tries). The nodes of the tries are labelled with the next hops according to prefixes in T. Some nodes u are also marked to record the fact that the path from the root to u stores a prefix of the table.

We can draw two cutlines on the trie, at levels 16 and 24. We obtain a set of at most 2^{16} sub-tries of height no more than $h = 8$. (We recall that the height is the numbering of levels in a trie, starting from 0 for the root.)

In order to analyze their common properties, we need to recall some terminology. Two tries are *isomorphic* if they have the same shape, the same labels, and the same marks on the nodes. Formally, two nodes u and v are isomorphic ($u \sim v$) if they are both null, or the following conditions hold: $label(u) = label(v)$, $mark(u) = mark(v)$, $left(u) \sim left(v)$, and $right(u) \sim right(v)$. Hence, two tries are isomorphic if and only if their roots u and v satisfy $u \sim v$. Note that we exploit this property in Section IV for keeping an auxiliary data structure for processing announcements and withdrawals.

For random data, we do not expect to find isomorphic sub-tries. There are at least 2^{300} sub-tries of height at most 8, since the number b_h of binary trees of height $h > 0$ is the solution to recurrence $b_h = b_{h-1}^2 + b_{h-1}(1 + \sqrt{4b_{h-1} - 3})$ as shown in [16], from which we can compute $b_h > 2^{300}$ for $h = 8$. If we account for the fact that our sub-tries have nodes labeled, the number is even larger. Hence the probability that two sub-tries are isomorphic, $p < 1/2^{300}$, is very near to zero. We can have 2^{16} such sub-tries for a routing table. Hence the probability that *no* two sub-tries are isomorphic is very near to one, i.e., $(1 - p)^{2^{16}} \approx 1$.

For IP lookups, we instead consider a weaker notion which is more relevant in our case. Given a trie of height h, let's expand it to its complete form (also called prefix expansion) so that all the leaves are on the same level. Nodes are still labeled and marked according to the prefixes in T, except that they are now part of a complete trie (which explictly represent all possible 2^h binary strings of length h). Note that each string is associated with its correct next hop when seen as part of an IP address.

We say that two tries of height h are *equivalent*, if the sequence of next hops in the leaves of the former is identical to that of the latter, when scanned in left-to-right order. In other words, when a lookup with h bits is performed on two equivalent tries, the next hops thus returned make them indistinguishable. Note that two isomorphic tries are equivalent while the reverse is not necessarily true, since different combinations of shapes and labels/marks can yield the same sequence of next hops.

We are therefore interested in selecting one representative for each class of equivalent tries. In our case, we apply this selection to sub-tries of height at most 8 obtained from the cutlines on levels 16 and 24 (corresponding to the middle-class prefixes). How many of them are equivalent? For random data, we expect that there are no equivalent sub-tries as the probability of finding two equivalent sub-tries is negligible. We can extend the above argument for isomorphic sub-tries to random sequences made up of 256 next hops.

Fortunately, we observe what we call the *middle-class effect* in real routing tables T when we build the trie on the prefixes in T:

> *Many sub-tries of height ≤ 8 on level 16 are equivalent with lots of repetitions, and they store the great majority of prefixes in T.*

So there is a good chance to store fewer than 2^{16} sub-tries by keeping just one representative for each equivalence class. Even though the majority of prefixes are middle-class (98% in our T), they do follow regular patterns in the routing table.

This fact is reinforced by observing that the empirical probability of finding that two consecutive sub-tries are equivalent is high, when scanning the sub-tries on level 16 in left-to-right order. For example in our table T, there are 13834 nonempty sub-tries of height at most 8 on level 16. We obtain just 5954 of them after removing a sub-trie if it is equivalent to its predecessor in a left-to-right scan (as we do during table construction). Among these, we are left with 3241 representatives of equivalence classes. These are not random data at all!

C. Two-layer approach

Following what claimed in the middle-class effect, we can transform the trie built on prefixes in T. We illustrate our approach by referring to T (shown in Table I). We first select only the prefixes of length up to 24 bits and the first 24 bits of longer prefixes, associating the dummy next hop with them. (We use the value of 255 in our experiments.) They form what we call *layer 1*. The set of remaining prefixes (with more than 24 bits) is augmented by taking their first 24 bits and associating with them the suitable next hop inherited from layer 1. All of these prefixes form *layer 2*. Table IV shows an example. Note that "dummy" prefixes of length 24 in layer 1 correspond to prefixes of length 24 with the correct next hop in layer 2. The number of such dummy prefixes cannot be larger than the number of prefixes longer than 24.

We then build a trie on the prefixes on layer 1 alone

layer 1		layer 2	
`65.10.10.0/24`	1	`192.168.0.0/24`	3
`192.168.0.0/17`	2	`192.168.0.0/32`	4
`192.168.0.0/18`	3	`192.168.0.0/29`	5
`192.168.64.0/18`	2		
`192.168.0.0/24`	255		

TABLE IV

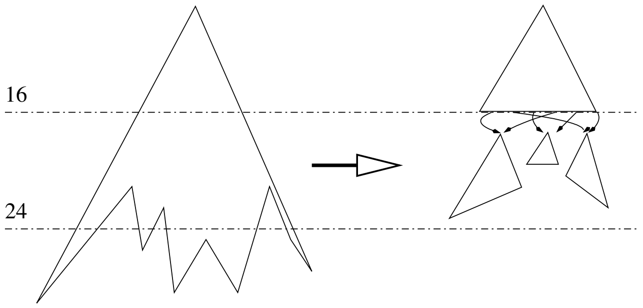

Fig. 3. Left: a trie for the prefixes in T. Right: the corresponding DAG in which the equivalent sub-tries of height at most 8 on level 16 are collapsed for the prefixes in layer 1.

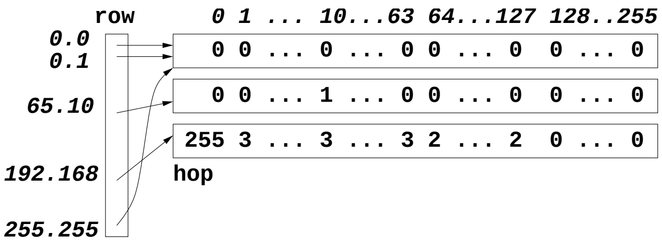

Fig. 4. The arrays row and hop for the prefixes in layer 1 shown in Table IV. No-route-to-host is the empty prefix with next hop 0.

and collapse equivalent sub-tries of height at most 8 on level 16, so as to form a direct acyclic graph (DAG), shown in Fig. 3. This graph gives a sufficiently good compression of the information stored in a routing table. As we shall see, the prefixes in layer 2 are small in number with respect to those in layer 1.

D. Lookup tables exploiting the middle-class effect

We now describe a simple, but powerful, lookup scheme based on the middle-class effect described in Section II-B and on the two-layer organization proposed in Section II-C.

Given our routing table T, we build two lookup tables for its prefixes. The first table stores the prefixes of layer 1 while the second table stores the prefixes of layer 2 (see again Table IV). We model our lookup scheme by these two layers. We begin by focusing on the lookup table for layer 1. (The lookup table for layer 2 depends on the implementation chosen as we shall see.)

We expand the upper part of the DAG in Fig. 3 that corresponds to the first 16 levels into a complete binary trie with 2^{16} leaves. The lower part of the DAG is a set of sub-tries of height at most 8, as previously mentioned. Using the definition of equivalence, we compute the sequence of 256 next hops obtained by each such sub-trie. We obtain a two-dimensional table for layer 1 as follows.

`hop`: This is the two-dimensional array of $\hat{\alpha} \times 256$ next hops, where $\hat{\alpha}$ is the number of non-equivalent sub-tries of height at most 8 on level 16 of the DAG, and each such sub-trie is represented by its sequence of $2^8 = 256$ next hops *without* RLE compression;

`row`: This is the array of 2^{16} entries mapping the first 16 bits of IP addresses to the suitable row of `hop`. (Equivalently, they represent the children pointers of DAG nodes on level 16.)

For example, with reference to layer 1 in Table IV, we obtain the lookup table shown in Fig. 4. Here, we have $\hat{\alpha} = 3$ rows in `hop`. Put into simple words, for any IPv4 address $x = x_1.x_2.x_3.x_4$, the next hop obtained by searching for x into the trie compactly represented by the DAG is that stored in $\mathtt{hop}[\,\mathtt{row}[x_1.x_2],\ x_3\,]$. So, an IP lookup for $x = \mathtt{192.168.32.27}$ successfully stops at layer 1 by returning the next hop 3, which is located at $\mathtt{hop}[\,\mathtt{row}[\mathtt{192.168}],\ \mathtt{32}\,]$. Instead, $x = \mathtt{192.168.0.27}$ requires a lookup in layer 2, since it returns the dummy value 255 stored in $\mathtt{hop}[\,\mathtt{row}[\mathtt{192.168}],\ \mathtt{0}\,]$.

Before discussing the experimental analysis on the lookup in Section III, we first assess the space occupancy of our scheme in the rest of this section.

Fact 1: Layer 1 occupies $\hat{\alpha} \times 256 + 2^{16} \cdot \#\mathit{pointer}$ bytes, where $\hat{\alpha} \le 2^{16}$ is the number of non-equivalent sub-tries of height at most 8 on level 16, and $\#\mathit{pointer} \ge (\log_2 \hat{\alpha})/8$ is the number of bytes encoding a pointer to `hop`'s rows.

In the worst case, `hop` occupies no more than 16 Mb and `row` needs 256 Kb (using 4-byte pointers) by Fact 1. This is actually a pessimistic estimate, since we only keep the sub-tries that are *not* equivalent each other. What we can experimentally observe is that our choice for representing layer 1 (which was data-driven) pays back in terms of space occupancy when compared to CDG.

In order to have a fair comparison with our scheme, we must add the space taken by the lookup table adopted for layer 2. We report in Table V the figures for several choices with router west.attcanada (see Section II-A), where we compare several methods for storing the prefixes in layer 2: CDG, array with binary search, k-way search (with $k = 8$ and $k = 2n$ where n is the number of prefixes), binary tries, and hybrid tries in which the first three levels are indexed by individual bytes and the next 8 levels (at most) are indexed by individual bits. Indeed, a lookup in layer 2 surely matches at least the first 24 bits by construction. Lookup times measure the number of microseconds for running 100,000 lookups.

We computed similar tables for other snapshots, as it turns out that hybrid tries are the best trade-off between space and lookup time. Choosing hybrid tries for storing

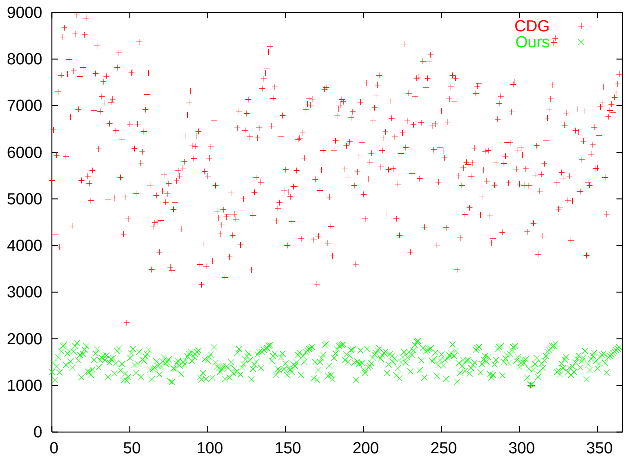

Fig. 5. Space occupancy of our scheme vs CDG for RIPE NCC. The x-axis reports the 365 daily snapshots of year 2003 and the y-axis the occupied space in bytes.

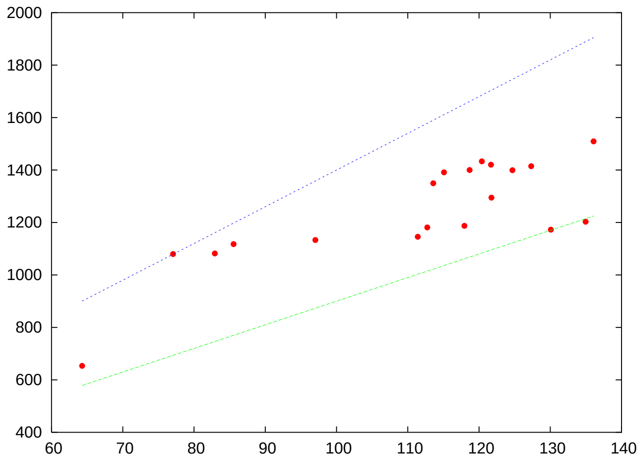

Fig. 6. Space occupancy of our scheme scales linearly with table size. The x-axis reports the number of prefixes and the y-axis the number of kilobytes taken. Plotted points are bounded by the two linear functions $f(n) = 9n$ and $g(n) = 14n$.

prefixes in layer 2, we report in Table VI the space improvement with respect to CDG for the 12 benchmark tables listed in Section II-A. As we can see, the column corresponding to our scheme gives a quite stable occupancy in space with respect to the routing table size (#prefixes). This is better highlighted if we consider the entire year 2003 of RIPE NCC, with the results for our scheme being plotted on the bottom of Figure 5.

The net result for our scheme is a lookup table whose space occupancy scales linearly with the number of prefixes. (Clearly, layer 1 alone scales as well; moreover, its maximum size is 16Mb.) Fig. 6 illustrates this behavior for the available monthly snapshots of RIPE NNC, from October 1999 to April 2004, with a number of prefixes ranging from 65841 (yielding $\hat{\alpha} = 1404$) to 138201 (yielding $\hat{\alpha} = 3241$). As can be noted, layer 1 has a size ranging in $[9n \dots 14n]$ bytes for n prefixes. For the sake of comparison, a straightforward storage of these prefixes alone in a routing table would require $6n$ bytes. In particular, each prefix requires a 4-byte word of memory; its prefix length and its next hop need one byte each.

	lookup time	Kb total	layer 1	layer 2
CDG	7012	2022	1521	501
Binary Search	5221	1556	1521	35
K Partition	5274	1556	1521	35
N Partition	5211	1608	1521	87
Binary Trie	5758	1649	1521	128
Hybrid Trie	5297	1649	1521	128

TABLE V

We also computed statistics for all daily snapshots of 2003 of RIPE NCC (see Section II-A). The total size of our lookup table (using a hybrid trie for layer 2) is in the range $[7n \dots 16n]$, thus confirming the linearity of space even in this case.

At this point, we may wonder whether a more sophisticated technique can better exploit the properties of the DAG in Fig. 3. For example, we could consider more cutlines and adaptive expansion of sub-tries [1]. While we do not claim this as a general rule, we believe that further improvement of the space occupancy of our scheme can significantly reduce the performance of lookup and update operations. As it becomes clear later, we want to easily update the data structure while guaranteeing very fast lookup operations. Our scheme is simple, very fast and keeps the space reasonable (although not at a minimum). Simplicity and efficiency are the major features of our approach. We give three

router	#prefixes	CDG (Kb)	ours (Kb)
aads	32505	3706	1084
att	121711	2188	1822
east.attcanada	127561	16418	1661
funet	41328	666	540
mae-west	71319	4643	1290
oregon-01	118190	9897	1596
oregon-03	142883	9026	2164
pacbell	45184	3170	982
paix	17766	2745	875
telstra	104096	8896	1490
telus	126687	11390	1724
west.attcanada	127576	16749	1664

TABLE VI

illustrative scenarios for implementing it; more are possible by varying the lookup scheme adopted for layer 2.

The first implementation uses SRAM with a uniprocessor, which is also the basis for our experiments (since it can be easily set up). We use hybrid tries for storing the long prefixes in layer 2. The size of our scheme for layer 1 is comparable to the current size of caches ($\approx$ 1–2Mb) according to our experiments. A random lookup accesses the table for layer 1 with a nearly 99.8% hit ratio, so that branch prediction works well for testing if lookup must go on querying layer 2. We report experimental data on this implementation in Section III.

The second implementation uses a bi-processor. One processor's cache holds layer 1 (the master), while the other processor's cache holds the hybrid trie for layer 2 (the slave). Lookups are in parallel but the slave processor can be interrupted when the master processor succeeds (which happens in the majority of cases).

The third implementation is challenging as it is purely hardware with a minimal requirement for logic. We store `row` into on-chip SRAM and `hop` into off-chip SRAM. We can preallocate the maximum size of both by Fact 1. We suggest using TCAM for layer 2, typically storing a few long prefixes (less than 15% in our data set). The expected size of the TCAM can be easily computed by performing statistics on the table prefixes longer than or equal to 24 bits. Again lookup is in parallel and can be implemented with negligible extra logic to select the output from TCAMs when the next hop in layer 1 is a dummy hop (255 with our data). We achieve one address lookup per clock cycle in this way.

E. Scaling to IPv6

Our solution has good chances to scale to IPv6 addressing. Although there is not so much available data, some downloadable routing tables are published in `http://net-stats.ipv6.cselt.it/bgp`. Here the relevant address type is global unicast. The first 64 bits are the most important ones for backbone routing, as the remaining 64 bits are for specifying an interface (e.g. a MAC address) where routing is mainly an intranet task. We also observe here the middle-class effect on a different scale. For our table, we have two cutlines at 24 and 48 bits and *no* prefixes are shorter than 24. We can blend our scheme and CDG by introducing a new array `col` and reducing the number of columns in `hop` with RLE in layer 1. Prefixes longer than 48 are stored in layer 2. For an address lookup, we hash the first 24 bits to a suitable entry of `row` and the next 24 bits to a suitable entry of `col`, which points to `hop`. If the returned hop is a dummy, we perform the lookup in layer 2 as before. Ultimately we just increase the number of memory accesses to 3 and require the computation of two hash functions. As a result, we expect that our method is competitive for IPv6 address lookup also, but we need more data to assess this rigorously.

III. Performing Lookups

The improved space bounds described in Section II makes our scheme more stable to use with respect to CDG. What about lookup time in IPv4? We recall that CDG requires 3 accesses in the worst case. We significantly improve this performance. We require just two accesses plus an access to layer 2, the latter with very low hit ratio (as we show next). As a result, our method is approximately 30% faster than CDG.

As previously mentioned, the lookup scheme is simple and requires trivial logic to be implemented in hardware. Assume that, for any given IP address $x = x_1.x_2.x_3.x_4$, we have the variable $\texttt{lx} = x_1.x_2$ storing the first 16 bits of x and $\texttt{rx} = x_3.x_4$ storing the last 16 bits, so that $x = \texttt{lx.rx}$. We use the right shift operator on `rx` to get byte x_3 and to perform a lookup. If we get the dummy value 255 in layer 1, we also need to perform a lookup in layer 2.

```
#define DUMMY  255
if ( (h1 = hop[ row[lx], rx>>8 ]) != DUMMY )
   return h1;
return lookup_layer2( lx.rx );
```

We measure the running time of our method and of CDG on the daily snapshots of RIPE NCC for the year 2003. We employ the synthetic traffic data for each individual snapshot as explained in Section II-A. As it can be noted in Fig. 7, our lookups are definitively faster than those in CDG by 30%. This is consistent with the fact that we reduce the number of memory accesses from 3 to 2.

It turns out that the role played by the data structures in layer 2 is rather limited in our data set, except for the single case that we discuss next. We report the experimental data in Table VII for the 12 benchmarks described in Section II-A. We use both random and synthetic data. For random data, the figures in italic correspond to random data employed in the experiments of [9], [13]. The columns hit-2 count how many hits our lookup made in layer 2. The other columns measure the running time in microseconds for 100,000 lookups.

We observe that the hit ratio for layer 2 is very low, so branch prediction in the if-statement works by returning the next hop `h1` of layer 1. As a result, our scheme essentially requires two memory accesses for a lookup. Note that, contrary to the rest of the snapshots in our data

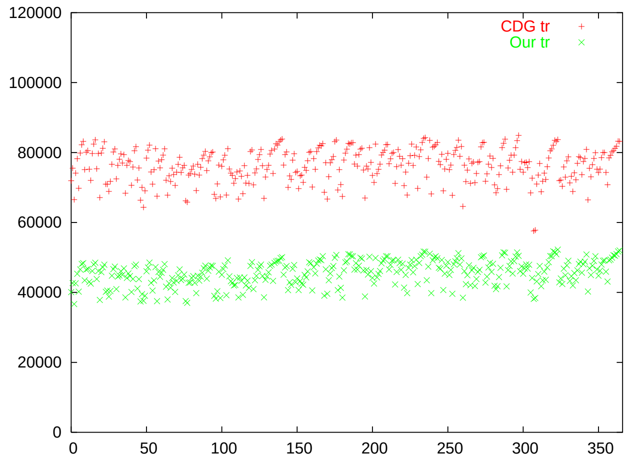

Fig. 7. Number of microseconds (on the y-axis) required by 1 million lookups in CDG (top) and in our scheme (bottom) using synthetic traffic. The x-axis reports the 365 daily snapshots of RIPE NCC 2003.

set, oregon-01 performs badly with respect to our scheme on the random data used in [9]. (On the other hand, it performs equally well with random data generated by us.) Here is a clear example showing that the choice of a hybrid trie as lookup mechanism in layer 2 is not enough powerful. Indeed, there are many prefixes of length between 28–32 in oregon-01, and lookups in layer 2 perform long matches, which is painful for trie searching.

If we use CDG for layer 2, we can get an improvement. This observation shows that the seemingly bad performance of our lookup scheme on oregon-01 is *not* due to layer 1 (which is quite stable and compact), but rather stems from layer 2. As we remarked before, in all other snapshots we observed a limited impact on the overall performance by the lookup method adopted in layer 2. Nevertheless, this limited impact appears not to be the case for the snapshot oregon-01.

A similar situation may occur if some malicious routing uses addresses that access layer 2 very often. We observe that the cache can adapt to this skewed access nicely since the number of routes in layer 2 is limited (see Section II-D) and most of the data structure for layer 2 is resident in the cache. To alleviate this problem, we exploit the fact that we definitively match the first 24 address bits in layer 2. We suggest using some a cache-efficient trie for layer 2 (see [17] for example).

We note that we obtain good performance in all other cases with just a hybrid trie on layer 2.

IV. Performing Updates

We now describe one of the main effects of our simplification of the lookup scheme. We show how to efficiently handle the updates of the lookup table when announcements and withdrawals of routes arrives on the fly. We do not need to rebuild the lookup table from scratch. Instead, we combine the best features of fast lookup using arrays with the flexibility of dynamically linked data structures while avoiding their drawbacks (rebuilding and slow lookup time, respectively).

We describe how to use our method (see Section II-D) by assuming that some reasonably efficient method has been adopted for layer 2 (e.g., tries, multi-level hashing, TCAMs, etc.). Again, we base our method on real data analysis to show that the great majority of updates involves layer 1, consistent with what was observed in the middle-class effect. We also make our scheme more robust by providing a good, exact upper bound on the number of entries changed in the lookup table in the worst case.

As described in Section II-D, we employ `hop` and `row` for layer 1. It is crucial to observe that `hop` is stored in *row-major* order. Since we adopt the maximum number of columns, 256, the only admissible size change in `hop` is to add or remove rows. Performing this change on the columns would result in a disaster, as the whole `hop` would need to be re-allocated dynamically, which can have a cost analogous to that of rebuilding. Here is why we opt for keeping all the 256 columns. We observe experimentally that using RLE on runs of equal next hops would reduce the number of columns by a negligible value at the price of reconstruction. So in this case, we prefer to have fast update and waste a bit of space. This also guarantees a high level of concurrent access to our lookup table during its lifetime.

We assume (realistically speaking) that the prefixes in route announcements and withdrawals are of length at least 8. (They can be shorter in case of heavy network

router	random			synthetic		
	CDG	ours	hit-2	CDG	ours	hit-2
RIPE NCC	10936	5922	1136	6970	4106	1074
aads	*7276*	*5903*	5463	7452	4775	4820
att	12605	7351	15	7872	4941	16
east.attcanada	15096	8429	3220	9164	5450	3116
funet	*3130*	*2461*	88	5036	2783	67
mae-west	*7217*	*5916*	2385	7425	4565	2401
oregon-01	*7740*	*9933*	11693	7265	6654	10651
oregon-03	14262	9529	3565	8790	6023	3525
pacbell	6126	5078	3899	6584	4233	3458
paix	*6306*	*5522*	9683	6934	4682	8703
telstra	*8468*	*7544*	3899	7966	5317	3690
telus	14011	8177	2095	8630	5279	2228
west.attcanada	15071	8353	3277	9167	5350	3050

TABLE VII

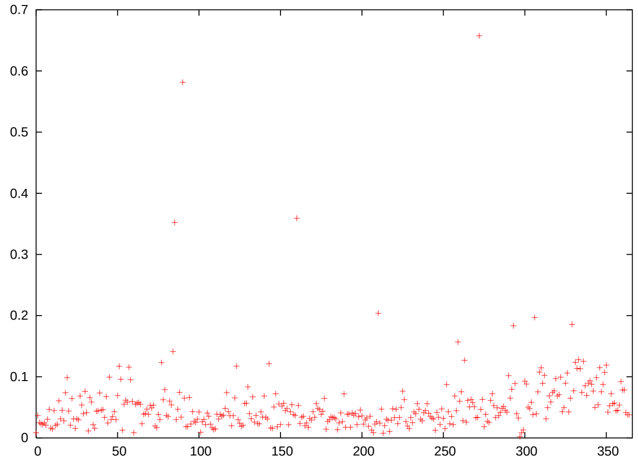

Fig. 8. On the y-axis, percentage of daily updates (less than 0.7%) involving layer 2 for RIPE NCC. The x-axis reports the 365 daily snapshots of year 2003.

failures, but then updating the routing table is a minor problem.) We also assume that there are at most 127 distinct next hop values in layer 1. We reserve the most significant bit in each entry of `hop` to mark it as a dummy. (Note that we do not use the dummy value of 255 anymore as in Section II-D.) Masking this bit yields the correct next hop value. If more than 127 values are needed, we add 32 bytes at the end of each row of `hop` to storing these mark bits. If more than 256 next hops are needed, we simply allocate two bytes per entry of `row`.

A. *Further data analysis*

We perform data analysis on the update traces for RIPE NCC. We collect the huge number of all the announcements and withdrawals available for year 2003 (see Section II-A). We report in Fig.8 the percentage of daily updates involving layer 2. Note that the maximum percentage is less then 0.7%, with almost all values below 0.1%. This confirms once again the middle-class effect that we observed on routing tables in Section II-B, motivating our choice to build layer 1 on the first 24 bits. Therefore, we suggest to use a well-tuned trie in layer 2, as its update cost does not significantly influence the overall performance of announcements and withdrawals in a router.

B. *Handling announcements and withdrawals*

We show how to efficiently process announcements and withdrawals that are produced during the execution of the border gateway protocol (BGP). When an announcements arrives, we have to insert a certain prefix p with its associated prefix length l_p and next hop h_p, into layer 1. Recall that $8 \leq l_p \leq 32$ by our assumptions. We distinguish among three main cases for describing the worst-case effect of this insertion on `row` and `hop`, illustrating them by using the example of layer 1 in Table IV and its associated arrays `row` and `hop` shown in Fig. 4. (We will discuss how to determine which entries change in Section IV-D.)

1) Case $l_p < 16$: Since $l_p \geq 8$, we have to change no more than 256 entries in `row`. However, each of them could change up to 256 entries in `hop`. The worst case is therefore that of changing $256 + 2^{16}$ entries. In practice, the number of entries is much smaller. In our example, if we announce route $p =$ `192.0.0.0` with prefix length $l_p = 8$ and next hop $h_p = 6$, we change the entries in `row`$[192.0 \ldots 192.255]$ except `row`$[192.168]$. All of them point to a new row in `hop` that is made up of all 6s. Note that we cannot create many such rows, as the number of distinct hop values is limited (to 127 in our case). In the row of `hop` pointed to by `row`$[192.168]$, we replace its right half of 0s with 6s. This is a situation that can be replicated over many rows, and that may cause the worst-case behavior, which we can bound as above.

2) Case $16 \leq l_p \leq 24$: This is the most frequent case according to the middle-class effect. We can change one entry of `row` to point from one row of `hop` to another, since the insertion of p needs to change some entries of the row previously pointed in that entry of `row`. We may need to add a new row when none of the existing ones match this change. In the worst case, we change no more than $1+256$ entries. Continuing our example, if we announce route $p =$ `192.168.128`, $l_p = 20$, and $h_p = 7$, we modify the row of `hop` pointed by `row`$[192.168]$, so that the 16 entries starting from position 128 change from 6s to 7s. Note that we do not need to create a new row as there is only one entry pointing. We should therefore know how many entries in `row` point a given row of `hop` to this end.

3) Case $l_p > 24$: We can change one entry in `row` and one in `hop`; however, the latter change may cause the creation of a new row in `hop` as discussed in case 2. Continuing our example, if we announce route $p =$ `192.168.128.12`, $l_p = 26$, and $h_p = 8$, we just have to change entry 128 in the row of `hop` pointed by `row`$[192.168]$. Its value changes from 7 to $128 + 7$ (we set the most significant bit to 1 to mark it as a dummy), and we must insert p, l_p, h_P into layer 2. Note that using 255 as a dummy value would also cause an insertion into layer 2 of `192.168.128.12/24` with next hop 7. This is a problem since we can change many entries in case 1, and this change can reflect on layer 2 as well. Our solution of using the most significant bit is simple since we do

not need to insert the first 24 bits of longer prefixes into layer 2 as previously illustrated in Table IV. This guarantees that an update falling into cases 1–2 does not propagate to layer 2 as a side effect.

Since we adopt a different encoding for dummy values, we need to make a slight change to the lookup procedure.

```
#define MSBIT 0x80
#define NO_ROUTE_TO_HOST  0
if ( ! ((h1 = hop[ row[lx], rx>>8 ]) & MSBIT) )
   return h1;
if ( (h2 = lookup_layer2( lx.rx )) != NO_ROUTE_TO_HOST )
   return h2;
return h1 & ~MSBIT;
```

If a lookup in layer 2 returns no-route-to-host, then we must return the next hop value (with its most significant bit cleaned) previously computed in layer 1. Although it may appear that we are harming the performance of the original lookup algorithm in Section III, we observe that the hit ratio for the first if-statement is very high and determines the real lookup cost, which stays unchanged according to the experimental evaluation discussed in Section III.

Withdrawals have an effect on `row` and `hop` similar to announcements, except that we have to handle "hidden" prefixes. When we delete a prefix, we should find the "parent" of that prefix and propagate its next hop downward to replace that of the deleted prefix. For example, starting from Fig. 4, the withdrawal of route 192.168.0.0/18 from layer 1 in Table IV causes the propagation of the next hop 2 (in place of 3), since it associated with the shorter prefix 192.168.0.0/17.

As a result we add or remove one row at most in `hop`. Removed rows are linked in a free list that can be reused for adding rows. This does not change the lookup procedure and its cost.

Since the main cost is given by the number of entries changed in `row` and `hop`, we computed statistics to account for this cost, classifying it according to cases 1–3 (both for announcements and withdrawals).

We processed the peak of Oct. 25–26, 2003, in router RIPE NCC. Table VIII shows that approximately 99.3% of the updates fall into case 2. Roughly half of them involve a prefix length $l_p = 24$, so they change just one entry in `hop`. Actually, the average number of changed entries in `row` and `hop` is nearly 1. For case 1, the most expensive one, the variance is high for a small number of updates while the rest of updates does not change any row of `hop`. On Oct. 25, just 1495 updates changed entries `row` and `hop`; on Oct. 26, there were 1889. These few updates changed between 100 and 1000 entries; we found a single example in which there were 20,985 changed entries, approaching the worst case.

The net result of the case analysis discussed so far is that updates are of bounded cost in layer 1, even in the worst case. This cost scales well with the number of updates and prefixes stored in layer 1.

Fact 2: In the worst case, the announcement or withdrawal of an IPv4 route changes at most 256 entries in `row` and at most 2^{16} entries in `hop` in case 1. The number of changed entries in `hop` becomes 256 in cases 2 and 3. In all cases, the empirical average number of changed entries is nearly 1.

C. Concurrent access

We have seen that, although rare, an update may change thousands of entries. Should we stop performing lookups meanwhile? Fortunately it is not so, as concurrent access is possible. It suffices that when a row is created in `hop`, the pointer in `row` is changed. Then, the row is correctly filled. In this way, any lookup accesses an entry of `hop` either before or after the update, but not during it! Concurrent access is possible in limited form also among updates, if they work on different rows of `hop`. We can safely guarantee the lookup functionality of our scheme while updating; so the cost of the update can be spread among a sequence of lookups without freezing the router for this reason (except for memory contention due to simultaneous access).

D. Auxiliary data structures for dynamic lookup table

We need to identify which entries change in arrays `row` and `hop` in order to handle announcements and withdrawals. There are several possibilities for this. We assume that the bookkeeping information is maintained elsewhere (e.g., see [2]) and does not interfere with the caching and prefetching of lookup data in `row` and `hop`. We propose one solution that seems reasonable to us. It makes use of a counter for each row of `hop` to count how many entries of `row` point to it. It also uses a hashing table for detecting equivalent sub-tries of height at most 8 on level 16, as they give rise to equal rows in `hop`. We propose to use fingerprints as hash functions, as they can be incrementally recomputed when only few entries change in a row.

We also need auxiliary data structures for quickly locating "hidden" prefixes. For example, in Table IV, prefix

date	#announce	#withdraw	case 1	case 2	case 3
10-25-04	20459780	139787	0.68%	99.31%	0.01%
10-26-04	11538757	144937	0.67%	99.30%	0.03%

TABLE VIII

`192.168.0.0/17` is hidden by `192.168.0.0/18` and `192.168.64.0/18`. However, if we withdraw `192.168.0.0/18`, then we must activate `192.168.0.0/17` and propagate its next hop. Another case occurs when two prefixes with the same next hop are one prefix of the other. If the shorter is deleted, then the longer emerges. We keep in a separate memory the DAG in Fig. 3 with the notable difference that isomorphic sub-tries are collapsed, instead of equivalent ones. Indeed, equivalent sub-tries cannot differentiate among the prefixes mentioned above, while isomorphic ones can.

V. Construction of the lookup table

The construction of our table consists in building a trie and then obtaining the DAG depicted in Fig. 3. It is worth noting that we insert the prefixes (truncated at 24 bits) into the trie in order of *nondecreasing prefix length*. If we do not follow this pattern, we have to propagate the next hop of the currently inserted prefix downward. In other words, we change the next hop to an already created set of nodes. If we follow the above pattern instead, we have to assign the next hop only to newly created nodes and this can happen once per node. This pattern gives a better performance in the worst case. For our tables, the most time-consuming construction was for oregon-03, in 365 milliseconds. Note that, since we can quickly handle updates, the construction time is less important than in static lookup tables.

VI. Related Work

Several approaches have been proposed in the last few years for the IP lookup problem. The survey in [1] describes the state of the art up to 2001, where CDG is shown to be an order of magnitude faster than its competitors. Since we improve over CDG, we claim that our method has a good performance by transitivity. More recent work is surveyed in [9] where recursive multibit tries (retries) are presented,which can also be applied to network clustering and telephone service marketing. We can obtain an indirect comparison with the retrie. To our knowledge, this is the most recent result that compares favorably with CDG for the IP lookup problem. It attains a variable improvement, which is mostly 30% as we do. It is based on dynamic programming and appears not to support quick updates. Other recent approaches are based on Bloom filters [2], multiple hashing [18], stratified trees [13], pipelined tries [11], [19], and biased skip lists [20], just to name a few. Several of them support updates and have small space requirements. It would be interesting to make a comparison with these methods, since our scheme does not require bit manipulation and hashing and makes two plain memory accesses most of the time.

References

[1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," in *IEEE Network*, 2001, pp. 8–23.

[2] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor, "Longest prefix matching using bloom filters," in *IEEE INFOCOM*, 2003, pp. 201–212.

[3] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 324–334, June 1999.

[4] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless inter-domain routing (CIDR): an address assignment and aggregation strategy," RFC 1519, 1993.

[5] J. Postel, "Internet protocol," RFC 791, 1981.

[6] S. Deering and R. Hinden, "Internet protocol, version 6 (IPv6)," RFC 1883, 1995.

[7] Chuck Semeria, "Understanding IP addressing: Everything you ever wanted to know," http://www.bergen.org/ATC/Course/InfoTech/Coolip/.

[8] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi, "IP address lookup made fast and simple," in *Proceedings of the 7th Annual European Symposium on Algorithms*, 1999, pp. 65–76.

[9] Adam L. Buchsbaum, Glenn S. Fowler, Balachannder Kirishnamurthy, Kiem-Phong Vo, and Jia Wang, "Fast prefix matching of bounded strings," *J. Exp. Algorithmics*, vol. 8, pp. 1–3, 2003.

[10] L. Rizzo, "Personal communication," 2003.

[11] Will Eatherton, George Varghese, and Zubin Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[12] Geoffrey Goodell, William Aiello, Timothy Griffin, John Ioannidis, Patrick McDaniel, and Aviel Rubin, "Working around BGP: An incremental approach to improving security and accuracy of interdomain routing," in *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, February 2003.

[13] Marco Pellegrini and Giordano Fusco, "Efficient IP table lookup via adaptive stratified trees with selective reconstructions," in *12th European Symposium on Algorithms*, 2004, pp. 24–35.

[14] M. Aida and T. Abe, "Pseudo-address generation algorithm of packet destinations for internet performance simulation," in *IEEE INFOCOM*, April 2001, pp. 1425–1433.

[15] Donald E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*, Addison-Wesley, Reading, MA, USA, second edition, 1998.

[16] A. V. Aho and N. J. A. Sloane, "Some doubly exponential sequences," *Fibonacci Quarterly*, pp. 429–437, November 1973.

[17] A. Acharya, H. Zhu, and K. Shen, "Adaptive algorithms for cache-efficient trie search," in *ALENEX*, 1999.

[18] Michael Mitzenmacher and Andrei Broder, "Using multiple hash functions to improve IP lookups," in *INFOCOM*, 2001, pp. 1454–1463.

[19] Anindya Basu and Girija Narlikar, "Fast incremental updates for pipelined forwarding engines," in *INFOCOM*, 2003.

[20] Funda Ergun, Suvo Mittra, Suleyman Cenk Sahinalp, Jonathan Sharp, and Rakesh K. Sinha, "A dynamic lookup scheme for bursty access patterns," in *INFOCOM*, 2001, pp. 1444–1453.