Theory Comput. Systems **38**, 481–501 (2005)
DOI: 10.1007/s00224-004-1183-1

Theory of Computing Systems

XOR-Based Schemes for Fast Parallel IP Lookups*

Giancarlo Bongiovanni[1] and Paolo Penna[2]

[1]Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza",
via Salaria 113, I-00133 Roma, Italy
bongio@dsi.uniroma1.it

[2]Dipartimento di Informatica ed Applicazioni "R.M. Capocelli", Università di Salerno,
via S. Allende 2, I-84081 Baronissi (SA), Italy
penna@dia.unisa.it

Abstract. An IP router must forward packets at gigabit speed in order to guarantee a good quality of service. Two important factors make this task a challenging problem: (i) for each packet, the longest matching prefix in the forwarding table must be quickly computed; (ii) the routing tables contain several thousands of entries and their size grows significantly every year. Because of this, parallel routers have been developed which use several processors to forward packets. In this work we present a novel algorithmic technique which, for the first time, exploits the parallelism of the router also to reduce the size of the routing table. Our method is scalable and requires only minimal additional hardware. Indeed, we prove that any IP routing table T can be split into two subtables T_1 and T_2 such that: (a) $|T_1|$ can be any positive integer $k \leq |T|$ and $|T_2| \leq |T| - k + 1$; (b) the two routing tables can be used separately by two processors so that the IP lookup on T is obtained by simply XOR-ing the IP lookup on the two tables. Our method is independent of the data structure used to implement the lookup search and it allows for a better use of the processors L2 cache. For real routers routing tables, we also show how to achieve simultaneously: (a) $|T_1|$ is roughly 7% of the original table T; (b) the lookup on table T_2 does not require the best matching prefix computation.

* The research by the second author was supported by the European Project IST-2001-33135, Critical Resource Sharing for Cooperation in Complex Systems (CRESCCO). Part of this work has been done while he was at the Mathematics Department, University of Rome "Tor Vergata".

1. Introduction

We consider the problem of forwarding packets in an Internet router (or backbone router): the router must decide the next hop of the packets based on their destinations and on its *routing table*. With the current technology which allows us to move a packet from the input interface to the output interface of a router [20], [18] at gigabit speed and the availability of high speed links based on optic fibers, the bottleneck in forwarding packets is the *IP lookup* operation, that is, the task of deciding the output interface corresponding to the next hop.

In the past this operation was performed by data link bridges [6]. Currently, Internet routers require the computation of the *longest matching prefix* of the destination address a. Indeed, in the early 1990s, because of the enormous increase of the number of endpoints, and the consequent increase of the routing tables, Classless Inter-Domain Routing (CIDR) and *address aggregation* had been introduced [7]. The basic idea is to aggregate all IP addresses corresponding to endpoints whose next hop is the same: it might be the case that all machines whose IP address starts by 255.128 have output interface $I1$; therefore we only need to keep, in the routing table, a single pair prefix/output $255.128.*.*/I1$: this is the conceptual format, as the actual format is something like 255.128.0.0/ 16 $I1$, plus additional information. Unfortunately, not all addresses with a common prefix correspond to the same "geographical" area: there might be so-called *exceptions*, like a subnet whose hosts have IP address starting by 255.128.128 and whose output interface is different, say $I2$. In this case we have both pairs in the routing table and the rule to forward a packet with address a is the following: if a is in the set[1] $255.128.*.*$, but *not* in $255.128.128.*$, then its next hop is $I1$; otherwise, if a is in the set $255.128.128.*$, then its next hop is $I2$. In general, the correct output interface is the one associated to the so-called *best matching prefix* $\mathsf{BMP}(a, T)$, that is, the longest prefix in T that is a prefix of a.

Even though other operations must be performed in order to forward a packet, the computation of the best matching prefix turns out to be the major and most computationally expensive task. Indeed, performing this task on low-cost workstations is considered a challenging problem which requires rather sophisticated *algorithmic solutions* [3], [5], [8], [11], [16], [22], [24]. Partially because of these difficulties, *parallel routers* have been developed which are equipped with several processors to process packets faster [20], [18], [15].

We first illustrate two simple algorithmic approaches to the problem and discuss why they are not feasible for IP lookup:

1. *Brute-force search on the table T*. We compare each entry of T and store the longest that is a prefix of the given address a.
2. *Prefix* (*re-*) *expansion*. We write down a new table containing all possible IP addresses of length 32 and the corresponding output interfaces.

Both approaches fail for different reasons. Typically, a routing table may contain several thousands of prefixes (e.g., the MaeEast router contains about 33,000 entries [13]),

[1] We consider a prefix $x*$ as the set of all possible string of length 32 whose prefix is x.

which makes the first approach too slow. On the other hand, the second approach would ensure that a single memory access is enough. Unfortunately, 2^{32} is a too large a number to fit in the memory cache, while access time in RAM is considered too large for high performances routers. Also, even a table with only IP addresses corresponding to end-points would suffer from the same problem: this is exactly a major reason why prefix aggregation is used!

In order to obtain a good tradeoff between memory size and number of memory accesses, a data structure named the *forwarding table* is constructed on the basis of the routing table T and then used for the IP lookup. For example, the forwarding table may consist of suitable hash functions. This approach, that works well in the case of searching for a key a in a dictionary T (i.e., the exact matching problem), has several drawbacks when applied to the IP lookup problem:

1. We do not know the length of the BMP. Therefore, we should try all possible lengths up to 32 for IPv4 [21] (128 for IPv6 [4]) and, for each length, apply a suitable hash function.
2. Even when an entry of T is a prefix of the packet address a, we are not sure that this one is the correct answer (i.e., the BMP). Indeed, the so-called *exceptions* require that the above approach must be performed for all lengths even when a prefix of a is found.

1.1. *Previous Solutions*

The above simple solution turns out to be inefficient for performing the IP lookup fast enough to guarantee millions of packets per second [15]. More sophisticated and efficient approaches have been introduced in several works in which a suitable data structure, named the *forwarding table*, is constructed from the routing table T [3], [5], [8], [11], [16], [22], [24]. For instance, in [24] a method ensuring $O(\log W)$ memory accesses has been presented, where W denotes the number of different prefix lengths occurring in the routing table T. This method has been improved in [22] using a technique called the *controlled prefix expansion*: prefixes of certain lengths are expanded thus reducing the value W to some W'. For instance, each prefix x of length 8 is replaced by $x \cdot 0$ and $x \cdot 1$ (both new prefixes have the same output interface of x). On one hand, the prefix expansion improves the number of memory accesses of the algorithm in [24]. On the other hand, its major drawback is the increase of the number of entries in the new table. This may lead to a forwarding table too large to fit in the L2 memory cache, thus resulting in a worse performance. Indeed, the main result of [22] is a method to pick a suitable set of prefix lengths so that (a) the overall data structure is not too big, and (b) the value of W' is as small as possible. Notice that an extreme solution would be to re-expand all prefixes up to its maximum length 32 and then construct one hash function for this new routing table. However, its size would be simply unfeasible even for a DRAM memory.

Actually, many existing works pursue a similar goal of obtaining an efficient data structure whose size fits into the L2 memory cache of a processor (i.e., about 1 Mb). This goal can be achieved only by considering real routing tables. For example, the solutions in [3], [5], [8], and [16] guarantee a constant number of memory accesses, while the size of the data structure is experimentally evaluated on real data; the latter affects the time efficiency of the solution.

These methods are designed to be implemented on a single processor of a router. Some routers exploit several processors by assigning different packets to different processors which perform the IP lookup operation using a suitable forwarding table. It is worth observing that:

1. All such methods suffer from the continuous growth of the routing tables [9], [2], [1]; if the size of the available L2 memory cache will not grow accordingly, the performance of such methods is destined to degrade.[2]
2. Other hardware-based solutions to the problem have been proposed (see [12], [19]), but they do not scale, thus becoming obsolete after a short time, and/or they turn out to be too expensive.
3. The solution adopted in [20] and [18] (see also [15]) exploits the parallelism in a rather simple way: many packets can be processed in parallel, but the time a single packet takes to be processed depends on the above solutions, which are still the bottleneck.

Finally, the issue of efficiently updating the forwarding table is also addressed in [11], [17], and [22]. Indeed, due to Internet routing instability [10], changes in the routing table occur every millisecond, thus requiring a very efficient method for updating the routing/forwarding table. Similar problems are considered in [14] for the task of constructing/updating the hash functions, which are a key ingredient used by several solutions.

1.2. *Our Contribution*

In this work we aim at exploiting the parallelism of routers in order to reduce the size of the routing tables. Indeed, a very first (inefficient) idea would be to take a routing table T and split it into two tables T_1 and T_2, each containing half of the entries of T. Then a packet is processed in *parallel* by two processors having in their memory (the forwarding table of) T_1 and T_2, respectively (see Figure 1). The final result is then obtained by combining via *hardware* the results of the IP lookup in T_1 and T_2.

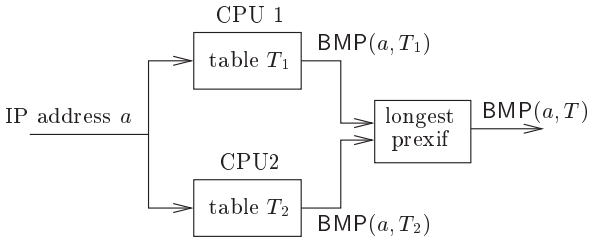

Fig. 1. A simple splitting of T into two tables T_1 and T_2 requires an additional hardware component to select the longest prefix.

[2] In our experiments we observed that the number of entries of a router can vary significantly from one day to the next one: for instance, the Paix router had about 87,000 entries on 1st November 2000, and about 22,000 only the day after.

The main benefit of this scheme relies on the fact that access operations on the L2 cache of the processor are much faster (up to seven times) than accesses on the RAM memory. Thus, working on smaller tables allows us to obtain much more efficient data structures and to face the problem of the continuous increase of the size of the tables [9]. Notice that this will not just increase the time a single packet takes to be processed once assigned to the processors, but also the throughput of the router: while our solution uses two processors to process one packet in one unit of time, a "classical" solution using two processors for two packets may take seven time units because of the size of the forwarding table.

Unfortunately, the use of the hardware for computing the final result may turn out to be infeasible or too expensive: this circuit should take in input $\mathsf{BMP}(a, T_1)$ and $\mathsf{BMP}(a, T_2)$ and return the longest string between these two (see Figure 1). An alternative would be to split T according to the leftmost bit: T_1 contains addresses starting by 0 (i.e., so-called CLASS A addresses) and T_2 those starting by 1. This, however, does not necessarily yield an even splitting of the original table, even when real data are considered [13].

The main contribution of this work is to provide a suitable way of splitting T into two tables T_1 and T_2 such that the two partial results can be combined in the simplest way: the XOR of the two sequences. This result is obtained via an efficient algorithm which, given a table T, for any positive integer $k \leq |T|$, finds a suitable subtable T_1 of size k with the property that

$$\textsc{Lookup}(a, T) = \textsc{Lookup}(a, T_1) \oplus \textsc{Lookup}(a, (T \setminus T_1) \cup \{\varepsilon\}),$$

where ε is the empty string and $\textsc{Lookup}(a, T)$ denotes the output interface corresponding to $\mathsf{BMP}(a, T)$, for any IP addresses a. Therefore, for every $k \leq |T|$, we can find two subtables of size k and $|T| - k + 1$, respectively.

The construction of T_1 is rather simple and the method yields different strategies which might be used to optimize other parameters of the two resulting routing tables. These, together with the guarantee that the size is smaller than the original one, might be used to enhance the performance of the forwarding table. Additionally, our approach is *scalable* in that T can be split into more than two subtables. Therefore, our method may yield a scalable solution alternative to the simple increase of the number of processors and/or the size of their L2 memory cache; for instance, rather than implementing a memory cache circuit of double size, we could simply double the number of processors and add a simple XOR circuit combined with our method. Notice that increasing the number of processors might be much simpler than implementing a memory cache of larger size and (approximatively) the same access time (e.g., the parallel routers in [20] and [18] use a rather large number of processors but each with memory cache of about 1–2 Mb). We believe that our novel technique may lead to a new family of parallel routers whose performance and costs are potentially better than those of the current solutions [20], [15], [17], [23].

We have tested our method with real data available at [13] for five routers: MaeEast, MaeWest, AADS, Paix, and PacBell. We present a further strategy yielding the following interesting performances:

1. A very small routing table T_1 whose size is very close to 7% of $|T|$. Indeed, in all our experiments it is always smaller but in one case (the Paix router) in which it

equals (i) 7.3% of $|T|$ when T contains over 87,000 entries, and (ii) 10.2% when $|T|$ is only about 6500 entries.
2. A "simple" routing table $T_2 = T \backslash T_1$ with the interesting feature that *no exceptions* occur, that is, every possible IP address a has at most one matching prefix in T_2.

So, for real data, we are able to circumscribe the problem of computing the best matching prefix to a very small set of prefixes. On one hand, we can apply one of the existing methods, like controlled prefix expansion [22], to table T_1: because of the very small size we could do this much more aggressively and get a significant speed-up. On the other hand, the way table T_2 should be used opens new research directions in that, to the best of our knowledge, the IP lookup problem with the restriction that no exceptions occur has never been considered before. Observe that table T_2 can be further split into subtables *without* using our method, since at most one of them contains a matching prefix.

Finally, we consider the issue of *updating* the routing/forwarding table, which any feasible solution for the IP lookup must take into account. We show that updates can be performed without introducing a significant overhead. Additionally, for the strategy presented in Section 3, all type of updates can be done with a constant number of operations, while keeping the structure optimality.

Roadmap. We describe our method and the main analytic results in Section 2. In Section 3 we present our experimental results on real routing tables. In Section 4 we conclude and describe the main open problems.

2. The General Method

In this section we describe our approach to obtain two subtables from a routing table T so that the computation of LOOKUP(a, T) can be performed in parallel with a minimal amount of additional hardware: the XOR of the two partial results.

Throughout the paper we make use of an equivalent representation of a routing table by means of trees. First consider the case in which the router has only two output interfaces, namely 0 and 1. In Figures 2 and 3 we show a routing table and the corresponding tree defined as follows:

1. Each vertex of the tree corresponds to a prefix in the routing table.
2. Each vertex has a label corresponding to the output value of the routing table, i.e., either 0 or 1.
3. Because of the best matching prefix rule, the labels of any two adjacent vertices are different, i.e., every path from a vertex to the root contains an alternated sequence of 0s and 1s.

In general, we consider a routing table $T = \{(s_1, o_1), \ldots, (s_n, o_n)\}$, where each pair (s_i, o_i) represents a prefix/output pair. Given two binary strings s_1 and s_2, we denote by $s_1 \prec s_2$ the fact that s_1 is a prefix of s_2. We can represent T as a *forest* (S, E) where the

Prefix	Output
100	0
1001	1
10001	1
100101	0
1001100	0
1001111	0
1001110	0

Fig. 2. A routing table.

set of vertices is $S = \{s_1, \ldots, s_n\}$ and for any two $s_1, s_2 \in S$, $(s_1, s_2) \in E$ if and only if

1. $s_1 \prec s_2$;
2. no $s \in S$ exists such that $s_1 \prec s \prec s_2$.

Finally, to every vertex s_i, we attach a label $0/1$ according to the corresponding output value of s_i in T.

In what follows we make use of this representation to derive a method to split T into subtables containing fewer elements than the original one. Moreover, to simplify the presentation, we assume that T always contains the empty string ε, thus making (S, E) a tree rooted at ε. Observe that this tree is not directly used to perform IP lookups. So, it will not be stored in the memory cache which will contain the forwarding tables derived from the subtables.

2.1. *The Main Idea*

Our method is based on the following idea: whenever a node $u \in T$ has the same label as its parent $p(u)$, then removing u from T and connecting its children to $p(u)$ does not change the result of $\text{LOOKUP}(a, T)$. Intuitively, $\mathsf{BMP}(a, T)$ is the lowest node $u \in T$ that matches with a. When we remove u from T the best matching prefix of a becomes node $p(u)$. So, if u and $p(u)$ have the same label, then $\text{LOOKUP}(a, T) = \text{LOOKUP}(a, \hat{T})$.

Now suppose that, in the tree T in Figure 3, we flip the label of vertex "1001" from 1 to 0 (this corresponds to changing the output value in the routing table). Then, using the above idea, it would be possible to *simplify the tree* (and hence the routing table) and obtain a tree $\hat{T}$ with only two entries: "100" and "10001" with labels equal to 0 and 1, respectively. Indeed, if $u = \mathsf{BMP}(a, T)$ is one of the nodes removed from T, then

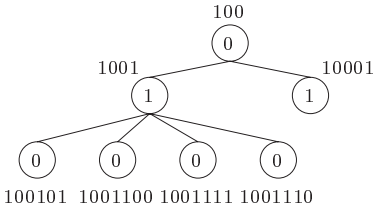

Fig. 3. An equivalent representation of the routing table in Figure 2.

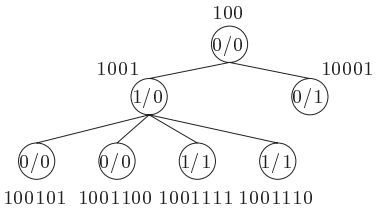

Fig. 4. The tree in Figure 3 with two new labels per vertex.

(i) node "100" also matches with a and (ii) the label of u and "100" are both equal to 0. Therefore, the value of LOOKUP(a, T) and LOOKUP$(a, \hat{T})$ is the same.

Unfortunately, we cannot simply flip some bits of the labels, since this would result in a loss of information and in an uncorrect lookup operation (e.g., according to routing table in Figures 2 and 3, all addresses "$1001100\cdots$" must be routed through output interface 1). However, we use the above idea to "spread" the vertices of the tree T into two different trees T_1 and T_2, each of them corresponding to a smaller routing table. Then the two routing tables can be used separately by two processors to compute the output interface value. Each packet is processed in *parallel* by two processors and the results are combined through a very simple Boolean circuit: the XOR of two bits.

We consider the example in Figure 4: the tree contains the same vertices (i.e., prefixes) as the tree in Figure 3, but each label (i.e., output interface) is replaced by a pair of labels with the property that *their XOR equals the old label* (see Figure 3). Intuitively, the two new labels represent the label of the vertex in the trees T_1 and T_2, respectively. We can then use the idea that *a vertex having the same label as its parent can be removed* to obtain the trees T_1 and T_2 in Figure 5.

2.2. *How to Split and Compact the Tables*

In what follows we describe in more detail our approach in the case of routers with any number of output interfaces.

2.2.1. *The split phase.* Given a routing table T, let T^{up} denote *any* subtree of T having the same root. Also, for any node $u \in T$, let $l(u)$ and $l'(u) = l_1(u)/l_2(u)$ denote its old

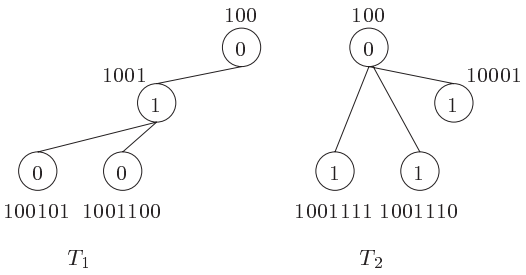

Fig. 5. The resulting two subtrees.

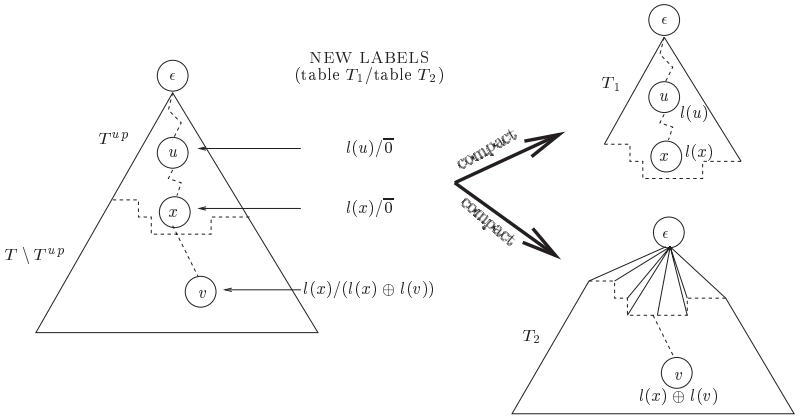

Fig. 6. An overview of our method: the subtree T^{up} corresponds to table T_1 after a compact operation is performed; similarly, $T \setminus T^{\text{up}}$ yields the table T_2.

and new labels, respectively. We assign the new labels as follows (see Figure 6):

- For any $u \in T^{\text{up}}$, $l'(u) = l(u)/\bar{0}$, where $\bar{0}$ denotes the bit sequence $(0, \ldots, 0)$.
- For any $v \in T \setminus T^{\text{up}}$, $l'(v) = l(x)/(l(x) \oplus l(v))$, where x is the lowest ancestor of v in T^{up}.

Let T' (respectively, T'') be the routing table obtained from T by replacing, for each $u \in T$, the label $l(u)$ with the label $l_1(u)$ (respectively, $l_2(u)$). It clearly holds that $l_1(u) \oplus l_2(u) = l(u)$. Hence,

$$\text{LOOKUP}(a, T') \oplus \text{LOOKUP}(a, T'') = \text{LOOKUP}(a, T).$$

2.2.2. *The compact phase.* The main idea behind the way we assign the labels is the following (see Figure 6):

1. All nodes in T^{up} have the second label equal to $\bar{0}$.
2. $T \setminus T^{\text{up}}$ contains upward paths where the first labels of each node are all the same.

Because of this, T_1 and T_2 contain redundant information and some entries (vertices) can be removed as follows. Given a table T, let $\text{COMPACT}(T)$ denote the table obtained by repeatedly performing the following transformation: for every node u with a child v having the same label, remove v and connect u to all the children of v. Then the following result holds:

Lemma 1. *Let* $T_1 = \text{COMPACT}(T')$ *and* $T_2 = \text{COMPACT}(T'')$. *Then* $|T_1| = |T^{\text{up}}|$ *and* $|T_2| = |T| - |T^{\text{up}}| + 1$.

Proof. We will show that no node in T^{up}, other than ε, will occur in T_2; similarly, no node in $T \setminus T^{\text{up}}$ will occur in T_1. Indeed, every node $u \in T^{\text{up}}$ has a label equal to $\bar{0}$ in T'' (see Figure 6). Since ε also has a label $\bar{0}$, $\text{COMPACT}(T'') = T_2$ will not contain any such u. Similarly, any node $v \in T \setminus T^{\text{up}}$ has its label in T' equal to some $l(x)$, where x is the

lowest ancestor of v in T^{up} (see Figure 6). Therefore, all nodes in the path from x to v have the same label $l(x)$ and thus will not occur in $\text{COMPACT}(T') = T_1$. This completes the proof. □

Lemma 2. *For any table T and for any address a, it holds that*

$$\text{LOOKUP}(a, T) = \text{LOOKUP}(a, \text{COMPACT}(T)).$$

Proof. Let $u_a = \mathsf{BMP}(a, T)$ and $v_a = \mathsf{BMP}(a, \text{COMPACT}(T))$. If $u_a = v_a$, then the lemma clearly follows. Otherwise, we first show that, in the tree T, either u_a is an ancestor of v_a or the other way around. By contradiction, let us assume that u_a and v_a are not an ancestor of one another, and let x be their lowest common ancestor. By definition, $x \prec u_a \prec a$ and $x \prec v_a \prec a$. Let u'_a and v'_a be the two children in the path from x down to u_a and v_a, respectively. By definition of T, it must hold that $u'_a \not\prec v'_a$ and $v'_a \not\prec u'_a$. Since $u_a \prec a$ and $v_a \prec a$, it holds that $u_a \prec v_a$ or $v_a \prec u_a$. We thus have two cases:

($u'_a \prec u_a \prec v_a$). By definition of v'_a, we also have $v'_a \prec v_a$, thus implying that either $u'_a \prec v'_a$ or $v'_a \prec u'_a$.

($v'_a \prec v_a \prec u_a$). By definition of u'_a, we also have $u'_a \prec u_a$, thus implying that either $u'_a \prec v'_a$ or $v'_a \prec u'_a$.

In both cases we have a contradiction with the fact that $u'_a \not\prec v'_a$ and $v'_a \not\prec u'_a$.

If u_a was an ancestor of v_a (i.e., $u_a \prec v_a$), then we would obtain $u_a = \mathsf{BMP}(a, T) \prec v_a \in T$. Since $v_a \prec a$, this would contradict the definition of BMP. We thus have that v_a is an ancestor of u_a in T (i.e., $v_a \prec u_a$). Since $\mathsf{BMP}(a, \text{COMPACT}(T)) = v_a \neq u_a$, it must then hold that $u_a \notin \text{COMPACT}(T)$. Moreover, in constructing $\text{COMPACT}(T)$, we have removed from T the node u_a and all of its ancestors up to v_a. This implies $l(u_a) = l(v_a)$, i.e., $\text{LOOKUP}(a, T) = \text{LOOKUP}(a, \text{COMPACT}(T))$. □

We have thus proved the following result:

Theorem 1. *For any routing table T and for any integer $1 \leq k \leq |T|$, there exist two routing tables T_1 and T_2 such that*

- $|T_1| \leq k$ *and* $|T_2| \leq |T| - k + 1$;
- $\text{LOOKUP}(a, T) = \text{LOOKUP}(a, T_1) \oplus \text{LOOKUP}(a, T_2)$, *for any address a.*

The above theorem guarantees that any table T can be divided into two tables T_1 and T_2 of size roughly $|T|/2$. By applying the above construction iteratively, the result generalizes to more than two subtables:

Corollary 1. *For any routing table T and for any integers $k_1, k_2, \ldots, k_l$, there exist $l + 1$ routing tables $T_1, T_2, \ldots, T_{l+1}$ such that*

- $|T_i| \leq k_i$, *for* $1 \leq i \leq l$;
- $|T_{l+1}| \leq |T| - k + l$, *where* $k = k_1 + k_2 + \cdots + k_l$;
- $\text{LOOKUP}(a, T) = \bigoplus_{i=1}^{l+1} \text{LOOKUP}(a, T_i)$, *for any address a.*

Remark 1. We mention that the bound on the overall size corresponding to all subtables is tight. Indeed, if the original table T does not contain any redundant information (i.e., $\text{COMPACT}(T) = T$), then every entry in $|T|$ must appear in one subtable. In other words, splitting T into $T_1, T_2, \ldots, T_{l+1}$ does not reduce the total number of entries.

Finally, we observe that the running time required for the construction of the two subtables depends on two factors: (a) the time needed to construct the tree corresponding to T; (b) the time required to compute T^{up}, given that tree.

While the latter depends on the strategy we adopt for T^{up} (see Section 3), the first step can always be performed efficiently. Indeed, by simply extending the partial order "$\prec$", a simple sorting algorithm yields the nodes of the tree in the same order as if we performed a BFS on the tree. Thus, the following result holds:

Theorem 2. *Let $t(|T|)$ denote the time needed for computing T^{up}, given the tree corresponding to a routing table T. Then the subtables T_1 and T_2 can be constructed in $O(|T| \log |T| + t(|T|))$ time.*

Also notice that if we want to obtain two subtables of roughly the same size, then a simple traversal (BFS or DFS) suffices, thus allowing us to construct the subtables in $O(|T| \log |T|)$ time. The same efficiency can also be achieved for a rather different strategy which we describe in Section 3.

2.3. *Updates*

In this section we show that, in several cases, our method does not introduce an overhead in the process of updating the forwarding table. We consider three types of updates: (a) label changes, (b) entry insertion, and (c) entry deletion. In particular, we assume that we have already computed the position, inside the tree T, of the newly added node or of the node to update. We describe a method to keep the two subtables T_1 and T_2 updated according to the change performed on the original table T. When performing these changes, we have to ensure that the following three properties are preserved by the new label pairs (see Figure 6): (**P1**) all nodes $u \in T^{\text{up}}$ agree on their second label (i.e., $l_2(u) = \bar{0}$), (**P2**) all nodes $v \in T \backslash T^{\text{up}}$, whose lowest ancestor in T^{up} is some node x, agree on their first label (i.e., $l_1(v) = l_1(x) = l(x)$), and (**P3**) for every node $p \in T$, $l_1(p) \oplus l_2(p) = l(p)$. Notice that, Properties (**P1**)–(**P1**) are defined as in Section 2.2.1 and are used in Section 2.2.2 to show that, by applying COMPACT to the two tables T' and T'', we obtain subtables T^{up} and $(T \backslash T^{\text{up}}) \cup \{\varepsilon\}$, respectively (see Figure 6 and Lemma 1). Our goal, however, is to avoid the recomputation of the subtables from scratch and to keep them updated.

In the remainder of this section, we make use of the following definition:

Definition 1. A node $x \in T^{\text{up}}$ is a *border* node if either (i) it is a leaf node in T or (ii) one of its children is in $T \backslash T^{\text{up}}$. A node $u \in T^{\text{up}}$ is an *internal* node if it is not a border node.

Informally speaking, border nodes are the "hard" case for the updates. In the example shown in Figure 7, we have $\{u, x\} \subseteq T^{\text{up}}$ and node x is a border node: every child v_i,

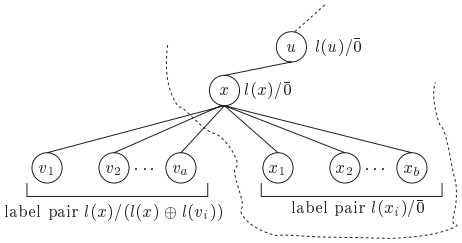

Fig. 7. The hard case for updates: if a border node x changes its label, then the label pairs of several of its children must also be updated.

with $i = 1, 2, \ldots, a$, is in $T \backslash T^{\mathrm{up}}$ and has the label pair $l'(v_i) = (l(x)/l(x) \oplus l(v_i))$. If the label of x changes from $l(x)$ to $\hat{l}(x)$ and we want to keep x in the set T^{up}, then its new label pair becomes $\hat{l}'(x) = (\hat{l}(x)/\bar{0})$. Unfortunately, this change implies that we should also change the label pair of every v_i into $\hat{l}'(v_i) = (\hat{l}(x)/\hat{l}(x) \oplus l(v_i))$. On the contrary, if we insist on keeping the first label of x equal to its old value $l(x)$, then its second label must be equal to $l(x) \oplus \hat{l}(x)$, so that the XOR of the two labels yields $\hat{l}(x)$. In this case every node x_i, for $i = 1, 2, \ldots, b$, will appear in both subtables T_1 and T_2. For b sufficiently large with respect to $|T|$, this results in a significant loss of the efficiency yielded by Theorem 1.

Since we do not want to recompute T_1 and T_2 from scratch, we make use of the following representation of table T. We consider T as a tree (as specified in Section 2) and, for each node u in T, we add the following fields: the label pair $l'(u) = (l_1(u)/l_2(u))$, as defined in Section 2.2.1 and a pointer $\mathsf{subtable}(u)$ to the entry of u in subtable T_i containing u, with $i \in \{1, 2\}$. We implement the tree by means of a list of children so that removing a node $u \in T$ and connecting all of its children to the parent p of u can be done in constant time.

Our goal is also to count how many updates we have to perform on each subtable. We thus assume that, for every update of a node $u \in T$, we also perform the analogous operation in the subtable T_i containing u. In particular, for every label change performed on label pair $l'(u)$, the corresponding update in T_i can be done in constant time using pointer $\mathsf{subtable}(u)$. Similarly, whenever we remove/add an entry from/to T^{up} (respectively, $T \backslash T^{\mathrm{up}}$), we can also remove/add that entry from the list representing T_1 (respectively, T_2) in constant time using pointer $\mathsf{subtable}(u)$.

In what follows we describe in more detail the updating procedures and their performances depending on the specific update to be performed. Notice that the time complexity is also an upper bound on the number of updates required on each subtable.

2.3.1. *Label changes.* Consider the situation in which the label of a prefix $p \in T$ changes from value $l(p)$ to a value $\hat{l}(p)$. We let $\hat{l}'(p) = (\hat{l}_1(p)/\hat{l}_2(p))$ denote the update of the pair $l'(p) = (l_1(p)/l_2(p))$ corresponding to this label change of p.

We distinguish three cases according to the left tree in Figure 6:

- p is an internal node of T^{up}, i.e., $p = u$ in Figure 6. This is the easy case, since it suffices to update the first label from $l(u)$ to $\hat{l}(u)$, that is, we set $\hat{l}'(u) =$

$(\hat{l}_1(u)/l_2(u)) = (\hat{l}_1(u)/\bar{0})$. Clearly, Properties (**P1**) and (**P3**) still hold. Moreover, Property (**P2**) is preserved since p is not a leaf node in T^{up}, thus implying that p cannot be the lowest ancestor, in T^{up}, of a node in $T \backslash T^{\text{up}}$.

- p is a border node of T^{up}, i.e., $p = x$ in Figure 6. As in the previous case, we have to change the first label from $l(u)$ to $l'(u)$; however, in this case p may be an ancestor of several other nodes in $T \backslash T^{\text{up}}$ (see Figure 7). For each node $v \in T \backslash T^{\text{up}}$, such that v is a descendant of x, we perform the following change: set $\hat{l}'(v) = (\hat{l}(x)/\hat{l}(x) \oplus l(v))$. By definition, after performing all these changes, the new labels of x and its descendants satisfy Property (**P2**). Moreover, Properties (**P1**) and (**P3**) clearly hold. This requires $O(|T_x|)$ steps, with T_x being the subtree of T rooted at x.
- p is a node in $T \backslash T^{\text{up}}$, i.e., $p = v$ in Figure 6. This is another simple case, since we only have to change the second label of v from $l(x) \oplus l(v)$ to $l(x) \oplus l'(v)$, that is, we set $\hat{l}'(v) = (l_1(x)/l_1(x) \oplus l'(v)) = (l(x)/l(x) \oplus l'(v))$. Properties (**P1**) and (**P3**) are clearly preserved. As for Property (**P2**), we observe that $l_1(v) = l(x)$ and any change of $l'(v)$ does not affect any node $w \in T \backslash T^{\text{up}}$: indeed, whether node w satisfies Property (**P2**) depends only on $l'(w)$ and on the labels of nodes in T^{up}. So, Property (**P2**) is also maintained.

It worth observing that in the first and in third cases, one label change in T translates into one label change in either T_1 or T_2.

Theorem 3. *Let* LABEL-CHANGE$(p, T, \hat{l}(p))$ *denote the label change procedure described above. The following results hold*: (i) *the resulting subtables* $\hat{T}_1$ *and* $\hat{T}_2$ *corresponding to the new label pairs satisfy* $\hat{T}_1 = T_1$ *and* $\hat{T}_2 = T_2$; (ii) *if* p *is not a border node*, *then* LABEL-CHANGE *runs in* $O(1)$ *time*; *otherwise*, *i.e.*, p *is a border node*, (iii) LABEL-CHANGE *runs in* $O(|T_p|)$ *time*, *where* T_p *denotes the subtree of* T *rooted at* p.

Finally, we mention that every label change could also imply some node deletion whenever the labels of two adjacent nodes become the same. This requires only an amount of time linear in the number of children of the updated node, and keeps the two subtables "simplified", without computing COMPACT(T') and/or COMPACT(T'') from scratch. Indeed, whenever a node $u \in T$ changes its label pair from $l'(u)$ to $\hat{l}'(u) = (\hat{l}_1(u)/\hat{l}_2(u))$, the following may happen:

1. $\hat{l}_1(u) = l_1(p)$ or $\hat{l}_2(u) = l_2(p)$, where $p(u)$ is the parent of u.
2. $\hat{l}_1(u) = l_1(v')$, for some child v' of u.
3. $\hat{l}_2(u) = l_2(v'')$, for some child v'' of u.

Notice that the above three cases are the only possible ones in which, because of the new label pair $\hat{l}'(u)$, two adjacent nodes in T' or in T'' have the same label. Therefore, in the worst case, we have to remove every v' (respectively, v'') of u from subtable T' (respectively, T''). Moreover, we may also have to remove node u from either T' or T'', if the first condition is met. Clearly, this can be done in time linear in the number of children of u.

2.3.2. *Insertion/deletion*

Insertion. Consider the situation in which a new node p must be inserted as a child of some existing node p' of T. In addition, adding entry p to table T may result in reconnecting some of the children of p'. In particular, let E_T denote the set of edges in T. Then all nodes in the set $\mathsf{children}(T, p) := \{u \in T \mid (p', u) \in E_T \wedge p \prec u\}$ must be connected to p and their labels may be also changed according to the labels of p. In the example shown in Figure 3, if p is the prefix '10011', then $p' = 1001$ and $\mathsf{children}(T, p) = \{1001100, 1001111, 1001110\}$.

In what follows we assume that we have inserted the new node p in the proper position in T and we have also reconnected all nodes in $\mathsf{children}(T, p)$. We next show how to update the labels in order to preserve Properties (**P1**)–(**P3**). Towards this end, we let $\hat{l}'(u) = (\hat{l}_1(u)/\hat{l}_2(u))$ denote the new label pair of a node $u \in T$ depending on the insertion of a new node p in T. We also let $\hat{l}'(p)$ denote the label pair of the newly added node.

We next show how to update the label pairs according to the following three cases (see Figure 6):

- p' is an internal node in T^{up}, i.e., $p' = u$ in Figure 6. We include p in T^{up} and we set $\hat{l}'(p) = (l(p)/\bar{0})$. Notice that since p' is an internal node, then all children of p' (if any) are in T^{up}. Therefore, any node $v \in \mathsf{children}(T, p)$ (if any) has the label $l'(v) = (l(v)/\bar{0})$, thus implying that Property (**P1**) is preserved. This also implies that p is not a border node and, thus, Property (**P2**) also holds (all labels of existing nodes in T are unchanged). Finally, Property (**P3**) clearly holds.
- p' is a border node in T^{up}, i.e., $p' = x$ in Figure 6. We perform this operation in two steps:
  1. We first insert node p assuming that its label is equal to the label $l(p')$ of its parent, that is, we define $l'(p) = (l(p')/\bar{0})$. Let $\hat{T}$ denote the resulting tree containing p with the label pair $l'(p)$.
  2. Then, in the tree $\hat{T}$, we update the label of p from $l(p')$ to $l(p)$ by invoking procedure LABEL-CHANGE$(p, \hat{T}, l(p))$ described in Section 2.3.1.

  Notice that, since in Step 2.3.2 we set $l'(p) = (l(p')/\bar{0})$, we have inserted p in T^{up}. Therefore, it might be the case that p becomes a border node. In this case, by Theorem 3, the running time of Step 2.3.2 is $O(T_p)$. Otherwise, i.e., p is not a border node, Theorem 3 implies that Step 2.3.2 requires $O(1)$ time.
- p' is a node in $T \backslash T^{\mathrm{up}}$, i.e., $p' = v$ in Figure 6. This is another easy case since we only have to set $\hat{l}'(p) = (l(x)/l(x) \oplus l(p))$, where x is the lowest ancestor of p' (and thus of p) in T^{up}. For every node $w \in \mathsf{children}(T, p)$ it holds that $l'(w) = (l_1(p')/l_1(p') \oplus l(w))$, where $l'(p') = (l(x)/l(x) \oplus l(p'))$, thus implying that Property (**P2**) is preserved. Finally, Properties (**P1**) and (**P3**) clearly hold.

Theorem 4. *Let* INSERT$(p, T, \hat{l}(p))$ *denote the insertion procedure described above, and let* $\hat{T}_1$ *and* $\hat{T}_2$ *denote the resulting subtables corresponding to the new label pairs. Then the following results hold*: (i) $|\hat{T}_1| + |\hat{T}_2| \leq |T| + 1$; (ii) $\hat{T}_1 \subseteq T_1 \cup \{p\}$; (iii) $\hat{T}_2 \subseteq T_2 \cup \{p\}$; (iv) *if* p *is not a border node, then* INSERT *runs in* $O(1)$ *time*; *otherwise, i.e.*,

p is a border node, (v) INSERT *runs in* $O(|T_p|)$ *time*, *where* T_p *denotes the subtree of* T *rooted at* p.

Deletion. Let $p \in T$ be the node to be removed from T, and let p' denote its parent. Removing node p causes the rearrangement of its children which, in the tree without T, become children of node p'. We define the new label pairs $\hat{l}'(u)$ in the same way as for the insertion.

We distinguish the following three cases:

- p is an internal node in T^{up}, i.e., $p = u$ in Figure 6. In this case, every child v of p is also in T^{up}, thus implying that $l'(v) = (l(v)/\bar{0}))$. Therefore, leaving all labels unchanged does not violate Property (**P1**). Also, Property (**P2**) holds since we did not change any label and we did not move any node from T^{up} into $T \backslash T^{\text{up}}$, or vice versa. Finally, Property (**P3**) follows from the fact that we did not change any label.
- p is a border node in T^{up}, i.e., $p = x$ in Figure 6. We perform this update in two steps:
  1. We first change the label of p from $l(p)$ to $l(p')$ by running LABEL-CHANGE$(p, T, l(p'))$. Let $\hat{T}$ denote the resulting tree containing p with the new label pairs.
  2. Remove node p from $\hat{T}$ and reconnect its children to the parent p' of p. Let $T_{\backslash\{P\}}$ denote the resulting tree.

  Notice that, since LABEL-CHANGE preserves Property (**P2**), in the tree $\hat{T}$, every descendant v of p belonging to $T \backslash T^{\text{up}}$ has the first label equal to $l(p')$. Therefore, after the removal of p from $\hat{T}$, Property (**P2**) holds. Also notice that Step 2.3.2 preserves Property (**P1**) since we do not change any label. Since Property (**P1**) is also preserved in Step 2.3.2, this property holds for the tree $T_{\backslash\{P\}}$. Similarly, Property (**P3**) is also maintained.
- p is a node in $T \backslash T^{\text{up}}$, i.e., $p = v$ in Figure 6. This is another easy case since we do not need any label change. Indeed, we have that $l'(p) = l'((l(x)/l(x) \oplus l(p)) = l'(p')$, where x is the lowest ancestor of p' (and thus of p) in T^{up}. For every node $w \in \mathsf{children}(T, p)$ it holds that $l'(w) = (l_1(p')/l_1(p') \oplus l(w))$, where $l'(p') = (l(x)/l(x) \oplus l(p'))$, thus implying that Property (**P2**) is preserved. Finally, Properties (**P1**) and (**P3**) clearly hold.

Theorem 5. *Let* DELETE(p, T) *denote the delete procedure described above, and let* $\hat{T}_1$ *and* $\hat{T}_2$ *denote the resulting subtables corresponding to the new label pairs. Then the following results hold*: (i) $|\hat{T}_1| + |\hat{T}_2| \leq |T| - 1$; (ii) $\hat{T}_1 \subseteq T_1$; (iii) $\hat{T}_2 \subseteq T_2$; (iv) *if p is not a border node, then* DELETE *runs in* $O(1)$ *time*; *otherwise, i.e., p is a border node*, (v) DELETE *runs in* $O(|T_p|)$ *time, where* T_p *denotes the subtree of* T *rooted at* p.

2.4. *Extensions of Our Method*

In this section we provide a more general view of our approach and motivate the use of XOR-based schemes (as opposed to other extensions/modifications of our approach using, e.g., the Boolean OR/AND operators).

Our choice of using the XOR operator "$\oplus$" is motivated by practical and theoretical considerations. From the practical point of view, as mentioned in Section 1, this operator determines the circuit complexity of the hardware that must combine the results of the two lookup operations (see Figure 1). Therefore, with "$\oplus$" we only need a few XOR gates. From a theoretical point of view, "$\oplus$" posses all the mathematical properties that are needed in order to achieve our main result (see Theorem 1). Indeed, we next provide a more general approach which uses a generic operator "$\odot$" with the following three properties: (i) associativity, (ii) identity element $i_\odot$, and (iii) inverse. We then show that these properties are necessary, since for both the OR "$+$" and the AND "$\cdot$" Boolean operators, we cannot achieve the same flexibility (notice that these operators do not have an inverse).

For the generalization of our approach, we refer to the description in Figure 6. In particular, the two new labels of each node are defined according to the following rules. Every node $x \in T^{\text{up}}$ has labels $l(x)/i_\odot$; a node $v \in T \backslash T^{\text{up}}$ has labels $l(x)/L[l(x), l(v)]$, with x being the lowest ancestor of v in T^{up} (see Figure 6) and with $L[\alpha, \beta] = \alpha^{-1} \odot \beta$. Doing so, we preserve the following properties used to achieve our results: (**P1**′) all nodes in T^{up} agree on their second label (i.e., $i_\odot$), (**P2**′) all nodes $v \in T \backslash T^{\text{up}}$, whose lowest ancestor in T^{up} is some node x, agree on their first label (i.e., $l(x)$), and (**P3**′) for every node, the "combination" of its two new labels gives the original one. In particular, let us observe that $l(x) \odot i_\odot = l(x)$ and

$$\begin{aligned} l(x) \odot L[l(x), l(v)] &= l(x) \odot (l(x)^{-1} \odot l(v)) = (l(x) \odot l(x)^{-1}) \odot l(v) \\ &= i_\odot \odot l(v) = l(v). \end{aligned} \tag{1}$$

Properties (**P1**′) and (**P2**′) imply that Lemma 1 still holds when replacing "$\oplus$" by "$\odot$", while Property (**P1**′) implies Lemma 2 with "$\odot$" in place of "$\oplus$". Putting things together, we can obtain the same as Theorem 1 and Corollary 1 for our generalized method.

Though the generalized approach does not give any improvement with respect to to the one with "$\oplus$", it provides a more abstract view of our method. We next show that properties (**P1**′)–(**P3**′) are somewhat necessary in order to obtain such a general result (namely, Theorem 1). Consider the case in which "$\oplus$" is replaced by the Boolean OR "$+$", and consider the example in Figure 3. We next show that it is impossible to have the splitting in Figure 5. Indeed, observe that, according to our approach, if the label is 1, then the new pair of labels must be $0/1$, $1/0$, or $1/1$, while a label 0 is always replaced by the pair $0/0$. In particular, all children of node "1001" in Figure 3 will have new labels $0/0$, thus implying that they must *all* appear in the same subtable (T^{up} or $T \backslash T^{\text{up}}$).[3] When considering the cases in which node "1001" has many children, it is impossible to obtain an even splitting of the original table using "$+$". A similar argument also applies to the AND operator "$\cdot$" simply by exchanging, in Figure 3, label values 0s and 1s.

[3] Things can be even worse if node "1001" gets the pair $1/1$, in which case all of its children appear in *both* subtables.

3. Experimental Results

These experiments have been performed on real routing tables of five routers: MaeEast, MaeWest, AADS, Paix, and PacBell. (Data available at [13].) In particular, we first observe that the tree T of the original table is a *shallow* tree, that is, its depth is always at most 6 (including the dummy node ε corresponding to the empty string). More importantly, the table *contains many leaf nodes*, i.e., nodes with no children. These nodes correspond to the entries u such that, for every $v \in T$, $u \not\prec v$. Therefore, if we restrict to this set of entries, we know that, for every IP address a, there is at most one matching prefix $u \prec a$, with u being a leaf node. In other words, in this case we do not need to find the best matching prefix. Based on this observation, we have tested the following strategy:

- The tree T^{up} contains all *nonleaf* nodes of T.

The idea is that of obtaining a table T_2 with *no exceptions* (i.e., all leaf nodes in T) and a table T_1 significantly smaller than $|T|$. Clearly, the smaller the size of T_1 the better:

- The small size of T_1 (with respect to the size of T) basically resolves the issue of the memory size of the data structure for the IP lookup in T_1.
- The particular structure of T_2 (i.e., no exceptions) may simplify significantly the problem and yield a data structure of smaller size (with respect to those solving the BMP problem).

It turns out that in all our experiments the size of T^{up} (and thus T_1) is always roughly 7% of $|T|$. Indeed, the only case in which it is larger than 7% is for the Paix router of 00/10/01: in this case, $|T_1|$ is about 7.3% of $|T|$ (see Table 1). Interestingly, the routing table of this router, for this day, has *over* 87, 000 entries, thus showing that our method is "robust" to size fluctuations (compare the same router of other days in Table 2).

We also emphasize that, very similar results have been obtained over both a period of one week (see Table 2) and over a sample consisting of snapshots of the same server for several months (see Table 1).

Table 1. Percentage of leaf nodes over one month (leaves/total entries, percentage).

Day (yy/mm/dd)	MaeEast	MaeWest	AADS	Paix	PacBell
00/10/01	22,462/24,018	30,195/32,259	27,112/28,820	80,812/87,125	34,266/36,313
	93.5%	93.6%	94%	92.7%	94.3%
00/11/01	21,935/23,507	31,115/33,249	28,239/30,002	9,922/10,620	21,900/23,273
	93.3%	93.5%	94.1%	93.4%	94.1%
00/12/01	23,575/25,187	29,862/31,927	27,158/28,852	9,851/10,522	20,763/22,093
	93.5%	93.5%	94.1%	93.6%	93.9%
01/01/01	23,140/24,760	29,680/31,752	26,906/28,562	10,421/11,092	36,788/39,088
	93.4%	93.4%	94.2%	93.9%	94.1%
01/02/01	23,152/24,753	29,019/30,993	26,255/27,833	10,776/11,454	37,925/40,320
	93.5%	93.6%	94.3%	94%	94%

Table 2. Percentage of leaf nodes over one week (leaves/total entries, percentage).

Day (yy/mm/dd)	MaeEast	MaeWest	AADS	Paix	PacBell
00/10/01	22,462/24,018 93.5%	30,195/32,259 93.6%	27,112/28,820 94%	80,812/87,125 92.7%	34,266/36,313 94.3%
00/10/02	22,380/23,932 93.5%	30,124/32,178 93.6%	27,066/28,755 94%	21,325/22,887 93%	34,446/36,511 94.3%
00/10/03	22,361/23,922 93.4%	30,038/32,094 93.5%	27,016/28,730 94%	80,776/87,100 92.7%	34,505/36,557 94.3%
00/10/04	22,426/23,991 93.4%	30,170/32,239 93.5%	27,121/28,832 94%	81,025/87,372 92.7%	34,315/36,387 94.3%
00/10/05	22,276/23,820 93.5%	30,249/32,320 93.5%	27,200/28,912 94%	81,030/87,374 92.7%	39,460/42,142 93.6%
00/10/06	22,252/23,800 93.4%	30,620/32,701 93.6%	27,945/29,763 93.8%	81,283/87,638 92.7%	34,465/36,535 94.3%
00/10/07	22,323/23,876 93.4%	30,414/32,488 93.6%	27,942/29,672 94.1%	81,179/87,542 92.7%	34,240/36,308 94.3%
00/10/08	22,339/23,902 93.4%	76,55/8,140 94%	28,000/29,734 94.1%	5,939/6,536 90.8%	7,824/8,275 94.5%

These two things together give strong evidence that this method guarantees the same performance over a long period of time (see also Table 3).

Justification. We next provide some evidence of the following two facts. First, with our method, the size of the table T_1 that contains the exceptions (i.e., two entries u and v with $u \prec v$) is quite small in spite of the fact that the original table T contains a larger percentage of exceptions. Moreover, the naive strategy of removing from T all suffixes and collecting them into table T_1 cannot achieve the same performances as our method.

Indeed, in Table 4 we report on two different measures: (i) the number of entries at a given height in the tree T: a node is at hight i if its distance from the root ε is i; (ii) the number of entries u whose subtree T_u has hight i. Observe that:

1. The height of an entry in the tree corresponds to the number of prefixes of such an entry. So, without considering node ε, the total number of entries of hight $i \geq 2$ yields the number of suffixes of other entries.
2. The number of entries u whose subtree T_u has hight 0 is equal to the number leaves in T.

Table 3. More results on the MaeWest for March 2002.

	Day						
	9th	10th	11th	12th	13th	14th	15th
Leaves	27,654	27,670	27,660	27,697	27,575	27,527	27,620
Total entries	29,635	29,648	29,633	29,686	29,542	29,485	29,585
Percentage	93.3%	93.3%	93.3%	93.2%	93.3%	93.3%	93.3%

Table 4. Number of entries with a given height vs number of entries whose subtree has a given height (MaeEast 00/10/01).

	Height						
	0	1	2	3	4	5	6
Entry	1	16917	6041	928	123	7	1
Subtree	22462	1362	162	27	3	1	1

Consider now the naive approach of defining subtable T_1 as the set of entries that are the prefix of other entries in T, that is, $T_1 := \{v \in T \mid \exists u \in T \backslash \{\varepsilon\} : u \prec v\}$ and $T_2 := T \backslash T_2$. By definition, table T_1 contains all entries at hight $i \geq 2$, while table T_2 those at height 1 or 0. This property ensures that no two entries u and u' in T_2, with $u \neq \varepsilon \neq u'$, satisfy $u \prec u'$. In other words, table T_2 contains no exceptions, while table T_1 does. Again, table T_1 represents the "hard" case as it requires the computation of the best matching prefix. Therefore, we would like its size to be as small as possible.

Unfortunately, for the real data in Table 4, the naive approach would give $|T_1| = 7100$ and $|T_2| = 16{,}918$, thus implying that T_1 contains roughly 29.5% of the total entries. With our method, instead, we obtain $|T_1| = 1556 + 1 = 1557$ and $|T_2| = 22{,}462$, with table T_1 containing only 6.5% of all entries in T. This justifies our approach as opposed to the naive one.

Updates. We briefly go back to the updating issue. Inserting a new node p as a child of some $v \in T \backslash T^{\text{up}}$ can be done by moving v in T^{up} and placing p in $T \backslash T^{\text{up}}$, without changing the labels of any other node. Every update involving a leaf node, i.e., a node $v \in T \backslash T^{\text{up}}$, can be performed with procedures INSERT and DELETE described in Section 2.3.2. Since v is not a border node, Theorems 3 and 5 imply that, in both cases, the running time is $O(1)$.

A label change of a border node node x (i.e., a leaf of T^{up} with at least one child in $T \backslash T^{\text{up}}$) can be performed by updating the labels of all children of x in T. Indeed, procedure LABEL-CHANGE requires changing the label pairs of all descendants of x that are in $T \backslash T^{\text{up}}$ (see Section 2.3.1). By definition of T^{up}, all descendants of a border node x must be children of x: indeed, every node in $T \backslash T^{\text{up}}$ is a leaf of T. Therefore, this operation requires an amount of time linear in the number of children of x.

We also mention that our approach may have a further benefit: assume that changes occur in the routing table with roughly a uniform distribution over all entries. Then we would expect many more changes in the "easy" subtable T_2 containing all leaf nodes. Therefore, the forwarding table corresponding to subtable T_1 can be more optimal for the lookup operation, rather than for the efficient update.

At the present state of our research we do not know whether the assumption on the uniform distribution of changes is realistic. We conjecture that a large fraction of all updates occur in the leaf nodes. Indeed, in Table 5 we show that the number of entries whose subtree has height 2 or more is almost the same over a period of one week (MaeEast router). This seems to denote the fact that changes occur more frequently for leaf nodes of T.

Table 5. The distribution of entries having a subtree of a given height (MaeEast router over one week).

Day (yy/mm/dd)	Height						
	0	1	2	3	4	5	6
00/10/01	22462	1362	162	27	3	1	1
00/10/02	22380	1359	162	26	3	1	1
00/10/03	22361	1366	163	27	3	1	1
00/10/04	22426	1370	162	28	3	1	1
00/10/05	22276	1352	162	25	3	1	1
00/10/06	22252	1356	161	26	3	1	1
00/10/07	22323	1362	161	25	3	1	1
00/10/08	22339	1369	163	26	3	1	1

4. Conclusion, Future Work, and Open Problems

We have introduced a general scheme which allows us to split a routing table into two (or more) routing tables T_1 and T_2, which can be used in parallel without introducing a significant hardware overhead. The method yields a family of possible ways to construct T_1: basically, all possible subtrees T^{up} as in Figure 6.

This will allow for a lot of flexibility. In particular, it might be interesting to investigate whether, for real data, it is possible to optimize other parameters. For instance, the worst-case time complexity of some solutions for the IP lookup [24], [22] depends on the number of different lengths occurring in the table. Is it possible to obtain two tables of roughly the same size and such that the set of prefix lengths is also spread between them?

Do strategies which split T into more than two tables have significant advantages in practice?

Finally, the main problem left open is that of designing an efficient forwarding table for the case of routing tables with no exceptions. Do any of the existing solutions get simpler or more efficient because of this?

Acknowledgments

We are grateful to Andrea Clementi, Pilu Crescenzi, and Giorgio Gambosi for several useful discussions. We also thank Pilu for providing us with part of the software used in [3] which is also used here to extract the information from the routing tables available at [13]. Our acknowledgments also go to Corrado Bellucci for implementing the strategy described in Section 3 and for performing some preliminary experiments. Finally, we thank the anonymous referees for suggesting we investigate the extensions of our method in Section 2.4, for reporting on a mistake on the statement of Lemma 1, and for several other insightful comments that greatly improved the paper.

References

[1] S. Bellovin, R. Bush, T.G. Griffin, and J. Rexford. Slowing Routing Table Growth by Filtering Based on Address Allocation Policies. Preprint available from http://www.research.att.com/~jrex/, June 2001.

[2] T. Bu, L. Gao, and D. Towsley. On Routing Table Growth. In *Proceedings of Globe Internet*, pages 2185–2189, 2002.

[3] P. Crescenzi, L. Dardini, and R. Grossi. IP Address Lookup Made Fast and Simple. In *Proc. of the 7th Annual European Symposium on Algorithms* (*ESA*), pages 65–76. Volume 1643 of LNCS. Springer, Berlin, 1999.
[4] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6). RFC 1883, 1995.
[5] M. Degernark, A. Brodnik, S. Carlesson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. *ACM Computer Communication Review*, 27(4):3–14, 1997.
[6] DIGITAL. *GIGAswitch/FDDI networking switch*. http://www.networks.europe.digital.com/html/products_guide/hp-swch3.html, 1995.
[7] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR) and Address Assignment and Aggregation Strategy. RFC 1519, September 1993.
[8] N. Huang, S. Zhao, and J. Pan C. Su. A Fast IP Routing Lookup Scheme for Gigabit Switching Routers. In *Proc. of the Annual Joint Conference of the IEEE Computer and Communications Societies* (*INFOCOM*), pages 1429–1436, 1999.
[9] G. Huston. Analyzing the Internet's BGP Routing Table. *The Internet Protocol Journal*, 4(1):2–15, 2001.
[10] C. Labovitz, G.R. Malan, and F. Jahanian. Origins of Internet Routing Instability. In *Proc. of the Annual Joint Conference of the IEEE Computer and Communications Societies* (*INFOCOM*), pages 218–226, 1999.
[11] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups Using Multi-Way and Multicolumn Search. *IEEE/ACM Transactions on Networking*, 7(3):324–334, 1998. Conference version in *Proc. IEEE INFOCOM* '98, pages 1188-1196.
[12] A. McAuley, P. Tsuchiya, and D. Wilson. Fast Multilevel Hierarchical Routing Table Using Content-Addressable Memory. US Patent Serial Number 034444, 1995.
[13] MERIT. IPMA statistics. ftp://ftp.merit.edu/ipma/routing_table, 2002.
[14] M. Mitzenmacher and A. Broder. Using Multiple Hash Functions to Improve IP Lookups . In *Proc. of the Annual Joint Conference of the IEEE Computer and Communications Societies* (*INFOCOM*), pages 1454–1463, 2001.
[15] P. Newman, G. Minshall, T. Lyon, and L. Huston. IP Switching and Gigabit Routers. *IEEE Communications Magazine*, 35(1):64–69, January 1997.
[16] S. Nilsson and G. Karlsson. Fast Address Look-Up for Internet Routers. In *Proc. of IEEE Broadband Communications*, pages 42–50, April 1998.
[17] D. Pao, C. Liu, A. Wu, L. Yeung, and K.S. Chan. Efficient Hardware Architecture for Fast IP Address Lookup. In *Proc. of the Annual Joint Conference of the IEEE Computer and Communications Societies* (*INFOCOM*), 2002.
[18] C. Partridge, P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W.C. Miller, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G.D. Troxel, and S. Winterble. A 50-Gb/s IP Router. *IEEE/ACM Transactions on Networking*, 6(3):237–247, 1998.
[19] T.-B. Pei and C. Zukowski. Putting routing tables into silicon. *IEEE Network*, 6(1):42–50, January 1992.
[20] Pluris Inc. Pluris Massively Parallel Routing. White Paper. http://www.pluris.com.
[21] J. Postel. Internet Protocol. RFC 791, 1981.
[22] V. Srinivasan and G. Varghese. Faster IP Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, 1999. Conference version in *Proc. ACM SIGMETRICS* '98, pages 1-10.
[23] D.E. Taylor, J.W. Lockwood, T.S. Sroull, J.S. Turner, and D.B. Parlour. Scalable IP Lookup for Programmable Routers. In *Proc. of the Annual Joint Conference of the IEEE Computer and Communications Societies* (*INFOCOM*), 2002.
[24] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proc. of the ACM Annual Conference of the Special Interest Group on Data Communication* (*SIGCOMM*), pages 25–36, September 1997.

Received June 24, 2003, *and in revised form December* 9, 2003, *and in final form January* 26, 2004.
Online publication February 24, 2005.