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 QoS. Two important factors make this task a challenging problem: (i) for each packet, the longest matching prefix in the forwarding table must be 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 to also reduce the size of the routing table. Our method is scalable and requires only a 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 $|T_2| \leq |T| - k$; (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 on 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.

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 to move a packet from the input interface to the output interface of a router [21,19] 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.

⋆ Research supported by the European project CRESCCO.

⋆⋆ Most of this work has been done while at the University of Rome "Tor Vergata", Math Department.

R. Petreschi et al. (Eds.): CIAC 2003, LNCS 2653, pp. 238–250, 2003.

In the past this operation was performed by data link Bridges [7]. 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 size of the routing tables, Classless Inter-Domain Routing (CIDR) and *address aggregation* have been introduced [8]. 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 $255.128.*.*/I1$. 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 $I2$. More in general, the correct output interface is the one of 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* [4,6,9,12,17,23,25]. Partially because of these difficulties, *parallel routers* have been developed which are equipped with several processors to process packets faster [21,19,16].

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 [14]), 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 number to fit in the DRAM. Also, even a table with only IP addresses corresponding to endpoints would be unfeasible: this is exactly a major reason why prefix aggregation has been introduced!

In order to obtain a good tradeoff between memory size and number of memory accesses, a data structure named *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 searching for a key a into a dictionary T (i.e., the exact matching problem), has several drawbacks when applied to the IP lookup problem:

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

1. We do not know the length of the BMP. Therefore, we should try all possible lengths up to 32 for IPv4 [22] (128 for IPv6 [5]) 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 [16]. More sophisticated and efficient approaches have been introduced in several works in which a suitable data structure, named *forwarding table*, is constructed from the routing table T [4,6,9,12,17,23,25]. For instance, in [25] 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 [23] using a technique called *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). The main result of [23] 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.

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 1Mb). This goal can be achieved only by considering real routing tables. For example, the solutions in [4,6,9,17] 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 [10,3,1]; 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 [13,20]), 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 [21,19] (see also [16]) 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.

[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 the 1st November 2000, and about 22,000 only the day after.

Finally, the issue of efficiently updating the forwarding table is also addressed in [12,18,23]. Indeed, due to Internet routing instability [11], 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 [15] 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 in exploiting the parallelism of routers with more than one processors 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 Fig. 1). The final result is then obtained by combining via *hardware* the results of the IP lookup in T_1 and T_2.

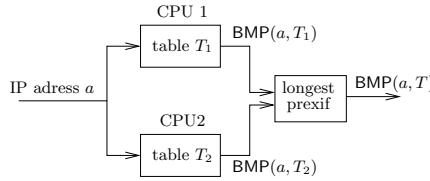

Fig. 1. A simple splitting of T into two tables T_1 and T_2 requires an additional hardware component to select the longest prefix.

The main benefit of this scheme relies in the fact that access operations on the L2 cache of the processor are much faster (up to seven times) than accesses on the DRAM memory.[3] Thus, working on smaller tables allows to obtain much more efficient data structures and to face the problem of the continuous increase of the size of the tables [10]. 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 1 packet in one unit of time, a "classical" solution using two processors for two packets may take 7 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 unfeasible 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 Fig. 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 [14].

[3] From now on we will improperly use the term DRAM also for SRAM adopted in some routers.

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

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

where $\textsc{Lookup}(a, T)$ denotes the output interface corresponding to $\mathsf{BMP}(a, T)$, for any IP addresses a.

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. We believe that our novel technique may lead to a new family of parallel routers whose performance and costs are potentially superior to those of the current solutions [21,16,18,24].

We have tested our method with real data available at [14] 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 to: (i) 7.3% of $|T|$ when T contains over 87,000 entries, and (ii) 10.2% when $|T|$ is only about 6,500 entries.
2. A "simple" routing table $T_2 = T \setminus T_1$ 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. By one hand, we can apply one of the existing methods, like controlled prefix expansion [23], to table T_1: because of the very small size we could do this much more aggressively and get a significant speed-up. By the other hand, the way table T_2 should be used opens new research directions in that, up to 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 Sect. 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 Sect. 2. In Sect. 3 we present our experimental results on real routing tables. In Sect. 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 $\textsc{Lookup}(a, T)$ 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. Let us 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 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 (i) $s_1 \prec s_2$, and (ii) no $s \in S$ exists such that $s_1 \prec s \prec s_2$. Finally, to every vertex s_i, we attach a label o_i according to the corresponding output interface.

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.

Our method consists of two phases which we describe below.

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 $l(u)$ denote its old label (i.e., its output interface in T) and let $l'(u) = l_1(u)/l_2(u)$ denote a pair of new labels. Intuitively, $l_i(u)$ represents the label of u in subtable T_i, $i = 1, 2$. We assign the new labels as follows (see Fig. 2):

- For any $u \in T^{up}$, $l'(u) = l(u)/\overline{0}$, where $\overline{0}$ denotes the bit sequence $(0, \ldots, 0)$;
- For any $v \in T \setminus T^{up}$, $l'(v) = l(x)/(l(x) \oplus l(v))$, 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,

$$\textsc{Lookup}(a, T') \oplus \textsc{Lookup}(a, T'') = \textsc{Lookup}(a, T).$$

The Compact Phase. The main idea behind the way we assign the new labels is the following (see Fig. 2):

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

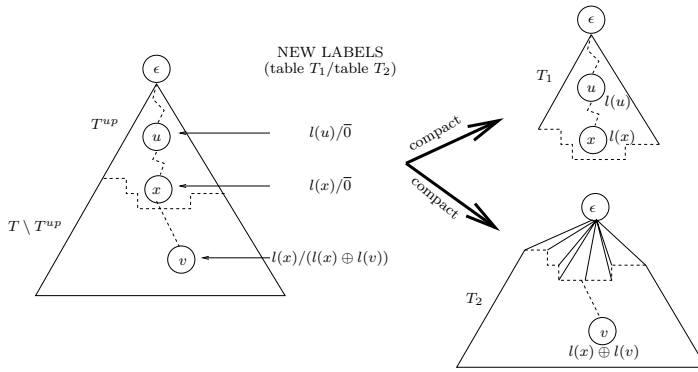

Fig. 2. An overview of our method: The subtree T^{up} corresponds to table T_1 after a compact operation is performed; similarly, $T \setminus T^{up}$ yields the table T_2.

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 $\textsc{Compact}(T)$ 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* $T_1 = \textsc{Compact}(T')$ *and* $T_2 = \textsc{Compact}(T'')$. *Then,* $|T_1| = |T^{up}|$ *and* $|T_2| = |T| - |T^{up}|$.

Proof. We will show that no node in T^{up}, other than ϵ, will occur in T_2; similarly, no node in $T \setminus T_1$ will occur in T_1. Indeed, every node $u \in T^{up}$ have label equal to $\overline{0}$ in T'' (see Fig. 2). Since also ϵ has label $\overline{0}$, $\textsc{Compact}(T'') = T_2$ will not contain any such u. Similarly, any node $v \in T \setminus T^{up}$ has its label in T' equal to some $l(x)$, where x is the lowest ancestor of v in T^{up} (see Fig. 2). Therefore, all nodes in the path from x to v have label the same label $l(x)$ and thus will not occur in $\textsc{Compact}(T') = T_1$. This completes the proof.

Lemma 2. *For any table* T *and for any address* a, *it holds that* $\textsc{Lookup}(a, T) = \textsc{Lookup}(a, \textsc{Compact}(T))$.

Proof. Let $u_a = \mathsf{BMP}(a, T)$ and $v_a = \mathsf{BMP}(a, \textsc{Compact}(T))$. If $u_a = v_a$, then the lemma clearly follows. Otherwise, we observe that, in constructing $\textsc{Compact}(T)$, 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., $\textsc{Lookup}(a, T) = \textsc{Lookup}(a, \textsc{Compact}(T))$.

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: (i)* $|T_1| \leq k$ *and* $|T_2| \leq |T| - k$, *and (ii) for any address* a, $\textsc{Lookup}(a, T) = \textsc{Lookup}(a, T_1) \oplus \textsc{Lookup}(a, T_2)$.

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

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 also Sect. 3), the first step can be always 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 perform 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 visit (BFS or DFS) suffices, thus allowing 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 Sect. 3.

2.1 Updates

In this section we show that our method does not yield an overhead in the process of updating the forwarding table. We consider two types of updates: (a) label changes, and (b) entry insertion/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.

Label Changes. Consider the situation in which the label of a prefix $p \in T$ changes from $l(u)$ to $l'(u)$. We distinguish three cases according to the left tree in Fig. 2:

- p is an internal node of T^{up}, i.e., $p = v$ in Fig. 2. This is the easy case, since it suffices to update the first label from $l(u)$ to $l(u)'$.
- p is a leaf of T^{up}, i.e., $p = x$ in Fig. 2. As above we have to change the first label from $l(u)$ to $l'(u)$; however, in this case p may be an ancestor of some other nodes in $T \setminus T^{up}$. This would require an update of all descendant nodes of p in T_2. We instead propose a simpler approach: move p from T^{up} into $T \setminus T^{up}$, so that the parent of p becomes a leaf of T^{up}. Even though this "lazy update" approach would make T_2 larger, this will happen only after several such updates. Therefore, we could periodically rebuild T^{up} so to maintain the two trees of roughly the same size.

– p is a node in $T \setminus T^{up}$, i.e., $p = v$ in Fig. 2. This is another simple case, since we only have to change the second label from $l(u) \oplus l(x)$ to $l'(u) \oplus l(x)$.

It worth observing that in the first and third case, one label change in T translates into one label change in either T_1 or T_2. In case p is on the "frontier" between T^{up} and $T \setminus T^{up}$, we can choose between performing one insertion/deletion (see next paragraph) or a certain number of updates in T_2. The choice of which to perform depends on several parameters, including how efficiently these operations are performed in the forwarding table.

Insertion/Deletion. Consider the situation in which a new node p must be inserted as child of some existing node of T. Again, we distinguish the following cases (see Fig. 2):

– p is child of an internal node $u \in T^{up}$. The new node can be simply inserted in its appropriate position with label pair $(l(p), \overline{0})$. As u is not a leaf of T^{up}, this will not affect any other node in $T \setminus T^{up}$; additionally, even if p may be inserted as parent of some node in T^{up} (a previously child of u), no further change is needed.
– p is a child a leaf node $x \in T^{up}$. We can deal with this case by first inserting p with the same label of x and then making a change of such label.
– p is a child of a node $v \in T \setminus T^{up}$. Another simple case since we only have to set the first label equal to $l(x)$ (see Fig. 2).

As for deletion of a node p, we only have to consider the case p is a leaf of T^{up}: we can simulate this by considering a label change of p so that its new label equals the label of its parent in T.

Finally, we mention that every label change could also imply some node deletion whenever the labels of two adjacent nodes become equal. This requires only a constant amount of time and keeps the two subtables "simplified", without computing COMPACT(T') and/or COMPACT(T'') from scratch.

3 Experimental Results

These experiments have been performed on real routing tables of five routers: Mae-East, Mae-West, AADS, Paix and PacBell. (Data available at [14].) 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., entries that have no suffix. Based on this, we have tested the following strategy:

– The tree T^{up} contains all *non leaf* nodes of T.

The idea is that of obtaining a table T_2 with *no exceptions* and a table T_1 of size significantly smaller than $|T|$. Clearly, the smaller the size of T_1 the better is:

- The small size of T_1 (w.r.t. the size of T) basically resolves the issue of the memory size of the 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 (w.r.t. 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 smaller than 7% is for the Paix router of 00/10/01. 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 1).

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

These two things together give a strong evidence that this method guarantees the same performance over a long period of time (see also. Table 2).

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

Day (yy/mm/dd)	**Mae-East**	**Mae-West**	**AADS**	**Paix**	**PacBell**
00/10/01	22462/24018 93.5%	30195/32259 93.6%	27112/28820 94%	80812/87125 92.7%	34266/36313 94.3%
00/10/02	22380/23932 93.5%	30124/32178 93.6%	27066/28755 94%	21325/22887 93%	34446/36511 94.3%
00/10/03	22361/23922 93.4%	30038/32094 93.5%	27016/28730 94%	80776/87100 92.7%	34505/36557 94.3%
00/10/04	22426/23991 93.4%	30170/32239 93.5%	27121/28832 94%	81025/87372 92.7%	34315/36387 94.3%
00/10/05	22276/23820 93.5%	30249/32320 93.5%	27200/28912 94%	81030/87374 92.7%	39460/42142 93.6%
00/10/06	22252/23800 93.4%	30620/32701 93.6%	27945/29763 93.8%	81283/87638 92.7%	34465/36535 94.3%
00/10/07	22323/23876 93.4%	30414/32488 93.6%	27942/29672 94.1%	81179/87542 92.7%	34240/36308 94.3%
00/10/08	22339/23902 93.4%	7655/8140 94%	28000/29734 94.1%	5939/6536 90.8%	7824/8275 94.5%

Table 2. More results on the Mae-West for March 2002.

Day	9th	10th	11th	12th	13th	14th	15th
Leaves	27654	27670	27660	27697	27575	27527	27620
Total entries	29635	29648	29633	29686	29542	29485	29585
Percentage	93.3%	93.3%	93.3%	93.2%	93.3%	93.3%	93.3%

Justification. It is worth observing that the high percentage of leaf nodes does *not* directly derives from the fact that the table T contains very few exceptions (i.e. suffixes) which is what we want to achieve in subtable T_2 (actually, we want no exceptions at all). Indeed, in the following table we compare the number of entries of a given height in the tree [4] vs the number of entries whose subtree has a given height:

Height	0	1	2	3	4	5	6
of entry	1	16917	6041	928	123	7	1
of subtree	22462	1362	162	27	3	1	1

Notice that, the percentage of nodes which are suffixes (i.e., height bigger than 1) is roughly 29.5% of the total entries; for the same table, the number of non-leaf nodes is 6.5% only. This implies that, in practice, our strategy performs much better than the intuitive method of collecting the exceptions into a table.

4 Conclusion, Future Work and Open Problems

We have introduced a general scheme which allows 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 Fig. 2.

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 [25,23] 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 and efficient forwarding table for the case of routing tables with no exceptions. Does 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 [4] which is also used here to extract the information from the routing tables available at [14]. Our acknowledgments also go to Corrado Bellucci for implementing the strategy described in Sect. 3 and for performing some preliminary experiments.

[4] The height of an entry in the tree corresponds to the number of prefixes of such an entry.

References

1. S. Bellovin, R. Bush, T.G. Griffin, and J. Rexford. *Slowing routing table growth by filtering based on address allocation policies.* http://www.research.att.com/∼jrex/, June 2001.
2. G. Bongiovanni and P. Penna. XOR-based schemes for fast parallel IP lookups. Technical report, University of Salerno, 2003. Electronically available at `http://www.dia.unisa.it/∼penna`.
3. T. Bu, L. Gao, and D. Towsley. On Routing Table Growth . In *Proceedings of Globe Internet*, 2002.
4. P. Crescenzi, L. Dardini, and R. Grossi. IP address lookup made fast and simple. In *Proc. 7th Annual European Symposium on Algorithms*, volume 1643 of *LNCS*, 1999.
5. S. Deering and R. Hinden. *Internet protocol, version 6 (IPv6).* RFC 1883, 1995.
6. 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.
7. DIGITAL, http://www.networks.europe.digital.com/html/products_guide/hp-swch3.html. *GIGAswitch/FDDI networking switch*, 1995.
8. V. Fuller, T. Li, J. Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): and address assignment and aggregation strategy.* RFC 1519, September 1993.
9. N. Huang, S. Zhao, and J. Pan C. Su. A Fast IP Routing Lookup Scheme for Gigabit Switching Routers. In *IEEE INFOCOM*, 2002.
10. G. Huston. Analyzing the Internet's BGP Routing Table. *The Internet Protocol Journal*, 4(1), 2001.
11. C. Labovitz, G.R. Malan, and F. Jahanian. Origins of Internet Routing Instability. In *IEEE INFOCOM*, 1999.
12. B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multi-way and Multicolumn Search. In *INFOCOM*, 1998.
13. A. McAuley, P. Tsuchiya, and D. Wilson. *Fast multilevel hierarchical routing table using content-adressable memory.* US Patent Serial Number 034444, 1995.
14. MERIT, ftp://ftp.merit.edu/ipma/routing_table. *IPMA statistices*, 2002.
15. M. Mitzenmacher and A. Broder. Using Multiple Hash Functions to Improve IP Lookups . In *IEEE INFOCOM*, 2001.
16. P. Newman, G. Minshall, T. Lyon, and L. Huston. IP Switching and Gigabit Routers. *IEEE Communications Magazine*, January 1997.
17. S. Nilsson and G. Karlsson. Fast address look-up for internet routers. *Proc. of ALEX*, pages 42–50, February 1998.
18. D. Pao, C. Liu, A. Wu, L. Yeung, and K.S. Chan. Efficient Hardware Architecture for Fast IP Adress Lookup. In *IEEE INFOCOM*, 2002.
19. 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.
20. T.-B. Pei and C. Zukowski. Putting routing tables into silicon. *IEEE Network*, January 1992.
21. Pluris Inc., White Paper, http://www.pluris.com. *Pluris Massively Parallel Routing.*
22. J. Postel. *J. Internet protocol.* RFC 791, 1981.

23. V. Srinivasan and G. Varghese. Faster IP Lookups using Controlled Prefix Expansion. In *Proc. of ACM SIGMETRICS (also in ACM TOCS 99)*, pages 1–10, September 1998.
24. D.E. Taylor, J.W. Lockwood, T.S. Sroull, J.S. Turner, and D.B. Parlour. Scalable IP Lookup for Programmable Routers. In *IEEE INFOCOM*, 2002.
25. M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proc. of ACM SIGCOMM*, pages 25–36, September 1997.