Binary Search Schemes for Fast IP Lookups

Pronita Mehrotra, Paul D. Franzon
ECE Department, North Carolina State University
Box 7911, Raleigh NC 27695
{pmehrot,paulf}@eos.ncsu.edu

Abstract—**Route lookup is becoming a very challenging problem due to the increasing size of routing tables. To determine the outgoing port for a given address, the longest matching prefix among all the prefixes, needs to be determined. This makes the task of searching in a large database quite difficult. Our paper describes binary search schemes that allow fast address lookups. Binary search can be performed on the number of entries or on the number of mutually disjoint prefixes. Lookups can be performed in O(N) time, where N is number of entries and the amount of memory required to store the binary database is also O(N). These schemes scale very well with both large databases and for longer addresses (as in IPv6).**

I. Introduction

Traffic on the Internet doubles every few months [1]. Additionally, the number of hosts on the Internet have been increasing steadily, forcing a transition from the 32 bit address of IPv4 to the 128 bit addressing scheme of IPv6[2]. With bandwidth hungry applications like video conferencing becoming increasingly popular, the demand on the performance of routers has become very high. The most time consuming part in the design of a router is the route lookup. This paper deals with route lookup schemes that allow faster lookups (a 10%-20% speed improvement) when compared to other comparable schemes.

When an IP router receives a packet on one of its input port, it has to decide the outgoing port packet depending on the destination address of the packet. To make this decision, it has to look into a large database of destination networks and hosts. Routing tables store only address prefixes which represent a group of addresses that can be reached from the output port. The problem of determining the next hop is equivalent to finding the *longest matching prefix*.

The rest of the paper is organized as follows. Section II discusses some of the previous work done in this field. Section III and Section IV then describe the details of our algorithms. We evaluated the performance of our schemes on practical routing tables, the results of which are presented in Section V and we finally conclude with Section VI.

II. Previous Work

Most of the approaches used to solve the longest matching prefix problem, fall under either the "thumb indexing" approach or binary search approaches [3]. The former approach does not scale very well with large address sizes (as in IPv6). Binary schemes on the other hand use searches on either the number of routing entries or the number of possible prefix lengths. These schemes would scale better with the higher address size of IPv6. Our paper deals with binary search methods where the search is performed on the routing table entries.

Gupta et al [4] presented a hardware implementation based on an indirect lookup scheme where the number of memory lookups required to determine the output port is quite small (1-2). Degermark et al [5] used compression techniques that allow a forwarding table to fit on on-chip caches. The hardware implementation by Huang et al [6] compacts a large forwarding table of 40,000 entries into 450-470KB. Still other schemes [7], [8] have used caching to improve the performance of route lookup. Caching relies on the temporal locality of data and this may not be very useful for core routers which exhibit very little temporal locality.

Binary tries use lesser memory to store the forwarding data bases, however the number of memory accesses required to evaluate the next hop is much higher. The NetBSD implementation uses a Patricia Trie [9], where a count of bits skipped is maintained for one-way branches. This reduces the average depth of the trie to some extent. The average length of the search in the Patricia implementation is 1.44 log(N) where N is the number of entries in the routing table. For large databases (>30,000), 22-23 memory accesses are still required. Level Compression (LC) can be used to further reduce the average depth of the trie [10].

Other approaches are variations of the binary search. Binary search by itself can work only with numbers. Prefixes in the routing tables represent range of numbers and so a straight implementation of binary search does not work. A few modifications to the binary search have been proposed. The first of these is the scheme by Lampson et al [11], where each prefix is expanded into two entries. The set of these entries is then processed to compute pointers to help in the search process. Yazdani et al [12] defined a sorting scheme to sort prefixes of different lengths and then applied binary search to the sorted list. However, their scheme leads to an unbalanced tree and a variable number of memory accesses. Waldvogel et al [13] suggested a hash based scheme where a binary search is performed on the number of possible prefix lengths. Their scheme scales well with the size of the routing table and at most 5 hash lookups (for IPv4) are required to determine the next hop address. However, as pointed out in [11], this scheme would not

TABLE I

A SAMPLE ROUTING TABLE WITH PREFIXES AND NEXT HOPS

Prefix	Next Hop
10*	3
1011*	9
011*	8
010110*	5
001*	4
101101*	2
011010*	6
011100*	1
10111*	8
00101*	7

TABLE II

PREFIXES FROM TABLE III AFTER SORTING

Prefix	Next Hop
00101*	7
001*	4
010110*	5
011010*	6
011100*	1
011*	8
101101*	2
10111*	8
1011*	9
10*	3

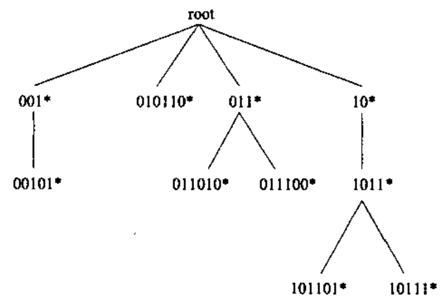

Fig. 1. Binary Tree Constructed from Table III

scale well with longer addresses as in IPv6.

Our work focuses on binary searches performed on the number of routing table entries. In that sense, it is closest to the work in [11] and their scheme (referred to as the LSV scheme) has been used to compare our results with, in the rest of the paper.

III. DESCRIPTION OF THE ALGORITHM

To understand the algorithm, we first look at a variable-degree tree constructed from the prefixes in Table III as shown in Fig. 1. To place prefixes in their relative positions in the tree, two conditions were used. Using the notation in [12], for two prefixes, $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$,

1) If $A \subset B$, then A is a parent of B (where the parent could be any node along the path from the node to the root of the tree)
2) If $A \not\subset B$ *and* $A < B$, then A lies on a subtree to the left of B. To compare A and B, if the prefix lengths of A and B are equal, i.e. $n = m$, then the prefixes can be compared by taking their numerical values. However, if $n \neq m$, then the longer prefix is chopped to the length of the shorter prefix and the numerical values compared.

By applying these conditions to all the prefixes, the tree in Fig. 1 can be constructed.

The problem with performing a binary search on variable length prefixes can now be seen. By simply sorting the prefixes in some fashion and performing a binary search, it would not be possible to determine the longest matching prefixes. For instance the prefixes could be sorted using the rules given above as shown in Table III. This is equivalent to having performed a post-order depth-first-search on the binary tree in Fig. 1.

Performing a simple binary search for a given address on the sorted prefixes, would not necessarily lead to the longest matching prefix. For instance, prefixes 101100* and 100* both lie between the entries 011* and 101101* but both of them have different longest matching prefixes. In general, the prefix could be any of the parent nodes or the root (default) node. Therefore, to determine the correct next hop, additional (and a variable number of) steps would need to be performed. This problem arose because the node 101101* did not carry any additional information about its parent nodes. We can avoid this problem by storing an additional field at all nodes that gives information about all the parent nodes. This additional field, which we call the Path Information Field, is a 32 bit entry (for IPv4) where a 1 in any bit position in the field means that there is a parent node with a prefix till that bit position. For example, for the leaf node of 101101* the Path Information field would look like 0...101010, i.e. the second bit (corresponding to 10*), the fourth bit (corresponding to 1011*) and the sixth bit (corresponding to the leaf node itself) are set to 1. In addition, the node would also contain a pointer to a list of next hop addresses for the corresponding bits in the path information field. In this case, the list of next hop addresses would be 2,9,3. Now by looking at the path information field the longest matching prefix can be determined and the correct next hop address obtained from the list. The data structure used at the nodes is shown in Fig. 2. In our implementation, we store the following information at each of the leaf nodes: the prefix, prefix mask, the next hop address corresponding to the leaf node, the number of internal nodes in the path and a pointer to the list of next hop addresses corresponding to the internal nodes. Table III shows the relevant information stored with each of the entries in Table III. Using the same example, if the address 101100* were to be searched for in the list, the search would point to between the upper and lower entries of 011* and 101101*, respectively. A match between the address 101100*

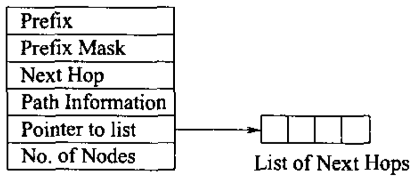

Fig. 2. Data Structure used at a Node

TABLE III

SEARCH SPACE OF THE PREFIXES IN TABLE III

Prefix	Next Hop	Path Information	Next Hop List
00101*	7	0...010100	4
001*	4	0...000100	-
010110*	5	0...100000	-
011010*	6	0...100100	8
011100*	1	0...100100	8
011*	8	0...000100	-
101101*	2	0...101010	9,3
10111*	8	0...011010	9,3
1011*	9	0...001010	3
10*	3	0...000010	-

and the lower entry 101101* results in a match of up to 5 bits. By looking at the path information field, bit 5 is not set to 1. The next lower bit that is set, is bit 4. Therefore, the longest matching prefix for the given address is 1011* and the corresponding next hop address in the next hop list is 9. In this case, the address needs to be compared with only the lower entry resulting from the search failure. To see why this is true, we look at different cases a search can end up in:

1) Between two leaf nodes with a common parent node. In this case either of the nodes can be used to determine the next hop address.
2) Between a node and its parent node. In this case, the parent node is the longest matching prefix and the parent node is the lower (larger) entry.
3) Between a node connected to the root node and a leaf node of the next branch (as in 011* and 101101*). In this case, nothing is gained by comparing with the smaller entry, since it can only lead to the root node (default next hop).

In all possible cases, it suffices to compare the address against only the lower entry to look for partial matches.

A. Building the Data Structure

Building the data structure is fairly simple and the steps involved are listed below:

Step 1: Each entry is read from the routing table and stored in an array.

Step 2: The entries are then sorted from the smallest to the highest. To compare prefixes, the rules listed previously are used. For prefixes of unequal lengths, if after chopping the longer prefix, the prefixes are equal then the shorter prefix is considered to be the larger of the two. Doing this ensures that in the following step, the child nodes get processed before the parent nodes.

Step 3: Entries from the sorted list are then processed and added in an array, one at a time. Each entry is also tested with the last array entry to see if it is a subset of the array entry.

Step 3a: If it is a subset, then the next hop information corresponding to the array entry is added to the next hop list of the array entry. The path information field of the array entry is updated and so is the field containing the number of nodes. This step is then repeated for previous entries till the test for subset fails. To see why this is necessary, consider the last prefix in Table III. When prefix 10* is compared with the last array entry (1011*), it is added in the next hop list of 1011*. However, 10* also lies in the path of 10111* and 101101* and the corresponding entries need to be updated as well.

B. Searching the Data Structure

To search for the longest matching prefix, a binary search is performed on the entries. The search algorithm is summarized below:

Step 1: Binary search of the address is performed on the array entries which leads to a value between two consecutive array entries.

Step 2: The address is then matched with the lower entry,and checked against the path information field. The longest matching prefix and the corresponding next hop address from the next hop list is picked. If no match is found, the default next hop address is returned.

C. Updating the Data Structure

Inserting or Deleting entries from the data space is equivalent to adding or deleting an entry from an array. To add an entry,a binary search is performed as outlined in the previous section, to find the location of the insertion. The entry is then added into the array, which is an O(N) process. The entry is also checked against entries above it to see if it is a subset or not, and the corresponding path information field and next hop list is updated. Deleting an entry follows a similar procedure. Updating the data structure, therefore, does not require the entire data structure to be built from scratch.

IV. USING DISJOINT PREFIXES FOR BINARY SEARCH

The search space used in the previous scheme can be reduced further by using only mutually disjoint prefixes. Two prefixes are considered disjoint, if none of them is a prefix of the other. It is easy to see that these correspond to the leaf nodes of the tree shown in Fig. 1. All internal nodes can be removed from the search space since the information corresponding to them is already contained in the path information field and the next hop list of the leaf nodes. The search space can then be shortened as shown in Table IV.

TABLE IV

SEARCH SPACE OF THE PREFIXES IN TABLE III

Prefix	Next Hop	Path Information	Next Hop List
00101*	7	0...010100	4
010110*	5	0...100000	-
011010*	6	0...100100	8
011100*	1	0...100100	8
101101*	2	0...101010	9,3
10111*	8	0...011010	9,3

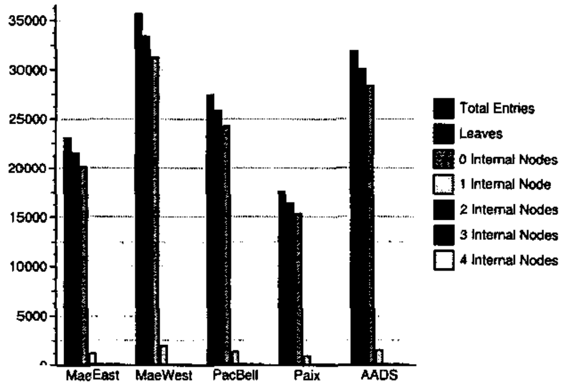

Fig. 3. Profile of routing tables from

From Table III and Table IV it might appear that a considerable amount of memory might be wasted in storing internal nodes a multiple number of times. For instance, next hop addresses corresponding to nodes 1011* and 10* are stored in the list of next hops for both 101101* and 10111*. This is not necessarily true. An examination of practical routing tables from [14] shows that most of the nodes in fact do not have any internal nodes and next hop lists to store. Fig. 3 shows the number of internal nodes for all the leaf nodes for various routing tables. From the figure, it can be seen that more than 93% of the leaf nodes do not have any internal nodes in the path to the root node. Therefore, the overhead in memory to store internal nodes multiple number of times is actually quite small.

The build and search algorithms need to be modified slightly to accommodate the changes. The main differences are that in the build phase, only the leaf nodes are added in the search space while the internal nodes only affect the path information field of the leaf nodes. In the search algorithm, the address needs to be matched against both the upper and lower entries to determine the better match. During updates, if the entry turns out to be an internal node, only the next hop lists and the parent information fields of the corresponding leaf nodes get updated.

TABLE V

AVERAGE SEARCH TIMES FOR DIFFERENT ROUTING TABLES

Routing Table	No of Entries	Binary Search (All nodes)	Binary Search (Only Leaves)	LSV Scheme
MaeEast	23113	662ns	610ns	761ns
MaeWest	35752	742ns	652ns	845ns
PacBell	27491	703ns	656ns	761ns
Paix	17641	640ns	634ns	739ns
AADS	31958	700ns	640ns	777ns

V. RESULTS AND DISCUSSIONS

We ran our algorithms on a Sun Ultra 5 with a 333 MHz processor and 512MB of RAM. The programs were written in C and compiled with gcc with the compiler optimization level 3. We also compared our results against the LSV scheme [11]. The binary search part of all algorithms were identical. We used the search time, build time and memory consumption to evaluate the performance of the schemes. Practical routing tables from [14] were used in the experiments.

Average Search Time Random IP addresses were generated and a lookup was performed using both schemes. Table V lists the average lookup times for different routing tables. From Table V it can be seen that our scheme which uses all nodes in the search space results in over 10% improvement in lookup speeds over the LSV scheme. With internal nodes eliminated from the search space, an improvement of 15-20% can be obtained.

The time taken to search for an entry is of the order of log(2N) where N is the number of entries in the routing table. Once the binary search is performed, an additional memory lookup is required (approximately half the time) to obtain the next hop address. In comparison, the search space in our schemes is $\leq N$. Our schemes therefore, result in 1-2 fewer memory accesses in just the binary search. After the search is narrowed down, our schemes, for most of the time does not require any additional memory lookup to determine the next hop address. For these reasons, the average lookup time in our scheme is lower than the lookup time in the LSV scheme.

Build Time of Data Structure The time required to build the searchable structure from the routing tables is shown in Table V. The time shown does not include the initial time taken to read entries from a file and store them in an array. This means that for all algorithms time starts from sorting the entries and ends when the searchable structure is built. As seen from the table the time taken to build the searchable space using LSV scheme is more than two times the time taken to build ours. A profile of the times taken, using gprof, shows that most of the difference can be accounted for by the initial sorting of entries. Since the number of entries in the LSV scheme is more than twice ours, sorting becomes a fairly expensive operation.

TABLE VI
TIME TAKEN TO BUILD SEARCHABLE STRUCTURE

Routing Table	No of Entries	Binary Search (All Nodes)	Binary Search (Only Leaves)	LSV Scheme
MaeEast	23113	80ms	80ms	210ms
MaeWest	35752	130ms	120ms	330ms
PacBell	27491	90ms	90ms	260ms
Paix	17641	60ms	50ms	150ms
AADS	31958	120ms	100ms	300ms

TABLE VII
MEMORY REQUIREMENT FOR DIFFERENT ROUTING TABLES

Routing Table	No of Entries	Binary Search (All Nodes)	Binary Search (Only Leaves)	LSV Scheme
MaeEast	23113	0.62MB	0.58MB	1.06MB
MaeWest	35752	0.96 MB	0.9MB	1.64MB
PacBell	27491	0.74MB	0.7MB	1.26MB
Paix	17641	0.48MB	0.45MB	0.81MB
AADS	31958	0.86MB	0.81MB	1.46MB

Build time for the scheme that uses only leaves in its searchable space is not significantly different from the one that uses all the nodes. This is due to the fact that most of the entries in these routing tables, end up as leaves and the overhead in adding internal nodes is very small.The build time will be particularly crucial for larger routing tables, because updating entries requires the entire searchable space to be built from scratch in the LSV scheme. For large routing tables (>100,000 entries), this could be serious problem in their scheme.

Memory Consumption Table V shows the memory required in storing the searchable structure for all schemes. Memory required for both binary schemes is close to half of that required for the LSV scheme. This is because each prefix in the LSV algorithm gives rise to two entries in the binary search table. Memory requirement for the scheme using only leaves in its searchable space is not significantly different from the one using all the nodes, for the reason pointed before.

One potential problem with our schemes is that the path information field has to be equal to the size of the address. For IPv4 this means that the path information field has to be 32 bits. For IPv6, this would mean storing a 128 bit field, leading to higher memory consumption and longer times to process instructions using this 128 bit field. In comparison, the additional information that LSV scheme uses are pointers, the size of which would depend on the number of entries. For a routing table of 100,000 entries, the pointers need to be only 18 bits wide. Storing the high and low pointers in their scheme, would make the extra memory consumption to 36 bits as compared to 128 bits in our case. However, this could be improved somewhat as we show next. If very long or very short prefixes do not exist in the routing tables, then the number of bits used in storing the path information can be reduced. For instance, for the MaeEast routing table, no prefixes exist for prefix lengths smaller than 8 bits and larger than 30 bits. The path information field, therefore, only needs to have 23 bits to store the relevant information. We do not have any data for how core routing tables would look like for IPv6, but we expect to be able to considerably reduce the number of bits used in storing the path information.

VI. CONCLUSIONS

We have described methods to adapt binary search to work with variable length prefixes. By storing information corresponding to parent nodes, we can reduce the time taken to search for a next hop address. Further improvement in performance can be obtained by using only mutually disjoint prefixes in the searchable space. 10-20% improvement in average lookup time over the LSV scheme can be obtained. In addition, reducing the search space to less than half, the amount of memory taken by the searchable space is also reduced to about half. The time required to build the searchable structure is similarly reduced considerably due to the fewer number of entries.

REFERENCES

[1] "Internet Host and Traffic Growth." (http://www.cs.columbia.edu/ hgs/internet/growth.html).

[2] "Internet Growth Summary." (http://www.mit.edu/people/mkgray/net/internet-growth-summary.html).

[3] N. McKeown and B. Prabhakar, "High Performance Switches and Routers: Theory and Practice," in *Hot Interconnects Tutorial Slides (http://tiny-tera.stanford.edu/ nickm/talks/index.html)*, Aug. 1999.

[4] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," in *Proc. IEEE INFOCOM'98*,(Sa n Francisco, CA), pp. 1382–1391, 1998.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, vol. 27, pp. 3–14, Oct. 1997.

[6] N.-F. Huang and S.-M. Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1093–1104, June 1999.

[7] T. cker Chiueh and P. Pradhan, "High-Performance IP Routing Table Lookup Using CPU Caching," in *Proc. IEEE INFOCOM'99*, pp. 1421–1428, 1999.

[8] T. cker Chiueh and P. Pradhan, "Cache Memory Design for Network Processors," in *Proceedings of Sxth International Symposium on High-Performance Computer Architecture, 2000*, vol. HPCA-6, pp. 409–418, 2000.

[9] K. Sklower, "A Tree-Based Routing Table for Berkeley Unix," in *Technical Report*,(U niversity of California, Berkeley).

[10] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1083–1092, June 1999.

[11] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," in *Proc. IEEE INFOCOM'98*, vol. 3, (San Francisco, CA), pp. 1248–1256, 1998.

[12] N. Yazdani and P. S. Min, "Fast and Scalabe schemes for the IP address Lookup Problem," in *Proc. IEEE Conference on High Performance Switching and Routing*, pp. 83–92, 2000.

[13] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proc. ACM SIGCOMM*, vol. 27, pp. 25–36, Oct. 1997.

[14] "Michigan University and Merit Network. Internet Performance Management and Analysis (IPMA) Project." (http://nic.merit.edu/ ipma).