Modified LC-Trie Based Efficient Routing Lookup

Ravikumar V.C Rabi Mahapatra J.C. Liu
Department of Computer Science, Texas A & M University
{vcr,rabi,jcliu}@cs.tamu.edu

Abstract

IP address lookup at the router is a complex problem. This has been primarily due to the increasing table sizes, growth in traffic rate and high link capacities. In this work we have proposed an algorithm for fast routing lookup with reduced memory utilization and access time. This approach shows significant performance improvement in the average case and optimizes the overall time taken for packet routing. Since storage requirement, processing time and number of lookups performed are reduced, power consumption by the router is also reduced. Our simulation result indicates that the proposed technique works approximately 4.11 times better than the standard LC Trie approach in the average case.

1. Introduction

Due to the technology advancement, the packet transmission rate is ever growing. With increasing number of hosts and sessions in the Internet, the processing of packets at the routers has become a major concern. While the current technology supports fast switching, packet forwarding is still a bottleneck. This is either due to a high processing cost or large memory requirement [9].

The routing tables implemented in most routers today generally follow the software-based approach, as they are more flexible and adaptive to any changes in the protocol. The hardware solutions are also not scalable and hence with the emergence of the IPv6, these approaches are nearly impractical. The software approaches expect to gain speed as the processing rate is doubled every 18 months (Moore's law).

From early 90's to late 90's, the number of entries in the lookup table has changed from linear to super linear. In 2002, the backbone router contains approximately 100,000 prefixes and is constantly growing [9]. A lookup engine deployed in the year 2002 should be able to support approximately 400,000-512,000 prefixes in order to be useful for atleast another 3 years. Thus the lookup algorithms must be able to accommodate future growth.

Apart from the lookup entries the speed of the links are doubling every year. The links running on OC768c (approximately 40Gbps) require the router to process 125 million packets per second (Mpps) (assuming minimum sized 40 byte TCP/IP packets) [9]. For applications that do not require quality of service, a lookup or classification algorithm that performs well in the average case is desirable. This is so because the average lookup performance can be much higher than the worst-case performance. For such applications the algorithm needs to process packets at the rate of 14.1 Mpps for OC768clinks, assuming an average Internet packet size of approximately 354 bytes [9].

2. Research Objectives

The main motivation behind our approach is as follows:

1) Reducing the lookup time: Many of the trie-based approaches like [1] [2] take 6-7 memory references for lookup, while hash technique combined with Tries in [3] takes 5 memory references. Our goal is to modify the trie-based approach to reduce the number of memory references for each lookup.

2) Reduce routing table storage: Implementing the routing table using trie [4] could be an expensive process. In this work our aim is to reduce the storage requirements by eliminating unnecessary and redundant data especially when the number of nodes are large.

3) Ability to handle large routing tables: Most of algorithms support routing table sizes for the current needs. However an algorithm needs to be scalable so that it supports the routing table storage requirements for atleast the next 3 years.

3. Previous work on lookup algorithms

We have classified the existing approaches into Trie and Non-Trie based approaches. We briefly analyze the different approaches with respect to lookup time and memory utilization.

Table 1. **Binary Strings to be stored in a Trie**

Nbr	String	Nbr	String
1	000	11	0110
2	00101	12	0111
3	010	13	10100
4	011	14	10101
5	100	15	10110
6	101	16	10111
7	110	17	11101000
8	1110100	18	11101001
9	0000	19	101000
10	0001	20	101010

3.1. Non-Trie based approaches

Linear search is simplest data-structure with the linked list of all prefixes in the forwarding table. Storage and time complexity is O (N). With the emergence of IPv6 this approach is almost impractical to continue. Binary search algorithms include the classical methods like the hashing and tree based approaches. These techniques cannot distinguish matches based on prefix length. However [3] takes care of this constraint and proposes a modified binary search technique that uses $\log_2$ (2N), where N is the number of routing table entries. These algorithms are also computation intensive and require a large storage especially as the lookup table grows. Caching is a better technique than other memory reference techniques, as it is faster [10] [6]. A fully associative memory, or content-addressable memory (CAM) [8] [9], can be used to perform an exact match search operation in hardware in a single clock cycle. CAM implementation is a costly mechanism and is not feasible. Thus the storage requirements for these become a limiting factor. Also CAM based approaches are for fixed length prefixes. A better solution is to use a ternary-CAM (TCAM), a more flexible type of CAM that enables comparisons of the input key with variable length elements.

3.2. Trie based approaches

Trie [4] is a general-purpose data structure for storing strings. Each string (prefix) in the routing table is represented by a leaf node in the trie. The longest prefix search operation on a given destination address starts from the root node of the trie [9]. Patricia trie is a modification of a regular trie. The difference is that it has no one-degree nodes. Each branch is compressed to a single node in a Patricia tree. Thus, the traversal algorithm may not necessarily inspect all bits of the address consecutively, skipping bits that formed part of the label of some previous trie branch. Patricia tree loses information while compressing chains; the bit-string represented by the other branches of the uncompressed chain is lost. This is overcome by a path compression technique that records by maintaining a skip value at each node that indicates how many bits have been skipped in the path [5]. The statistical property of this trie (Patricia trie) indicates that it gives an asymptotic reduction of the average depth for a large class distribution [11]. However when the trie is densely distributed this approach fails in terms of storage and processing for lookup. Level compressed trie (LC-trie) [1] is a modification to this scheme for densely populated tries. An *LC-trie* is created by path compressing a binary trie and expanding every node rooted at a complete sub-trie of maximum depth to create a 2^k-degree node [9]. This expansion is done recursively on each subtrie of the trie. It replaces the *i* highest complete levels of the binary trie with a single node of degree 2^i. All the internal nodes represented in this trie contain no information but pointers to the first child. Hence a separate vector is needed to store all possible prefix in a *prefix vector* in case of a failure in search. Also since each node is traversed again by backtracking in case of failure this is an inefficient method both in terms of processing time and storage. Thus all the variants of the trie-based approach are not storage efficient and require a lot of processing.

4. Modified LC-Trie

In this section we describe a scalable time efficient level compression technique. The approach presented here is motivated to minimize the access time and memory utilization during routing table lookup, to transfer packets to the appropriate Ethernet port. A few approaches like [1] have been proposed to better storage and time complexity. Our approach is based on the LC-trie approach for compressing the trie. We use the same compression technique as in LC-trie. However we try to avoid additional storage (LC-Trie uses additional data structures like base, prefix and nexthop vector to store prefixes) and processing (like backtracking), which are the major flaws of LC-Trie approach. Figure 1 represents the tree corresponding to Table 1 for the proposed approach. From the Figure 1 we see

that unlike the LC-trie approach our approach stores all the prefixes either in the internal nodes or leaf nodes.

The algorithm first converts the routing table entries into a binary trie. Then path compression is done on this trie to reduce its depth. This path-compressed trie is now level compressed by storing the sub strings at the internal nodes and the strings at the leaves. When the routing table is built we use a FILL FACTOR (this represents the maximum number of branches that each node can have during the build) to help make the future updates easier.

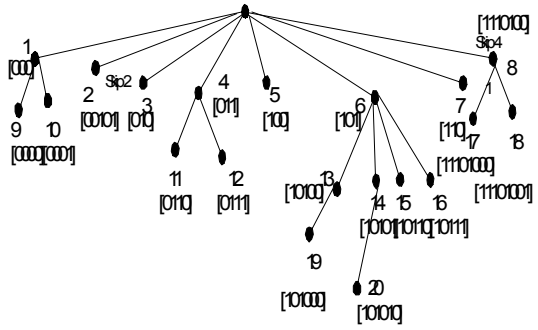

Figure 1. Scalable Time Efficient Level Compression

4.1 Storage data structure

Each node of our trie is represented as in Figure 2. Following is a brief description of the significance of each of the fields in the data structure.

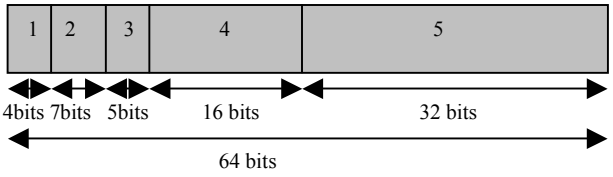

Figure 2. Proposed Data structure for node in routing table (IPv4)

Branching factor [0:3]: This indicates the number of descendents of a node. This is a 4-bit value and a maximum of 16 branches to a single node can be represented.

Skip value [4:10]: This indicates the number of bits that can be skipped in an operation. This is a 7-bit value and a maximum of 128-bit skip can be represented.

Port [11:15]: This represents the output port for the current node in case of a match. This is a 5-bit value and this field can represent a maximum of 32 output ports.

Pointer [16:31]: This is a pointer to the leftmost child in case of an internal node and NULL in case of a leaf node. This is a 16 bit value and can represent a maximum of 65536 prefixes. The current implementation assumes the number of routing table entries to be less than 65536. However a scalable solution to consider more than 5,000,000 prefixes is discussed later.

String [32:63]: This represents the actual value of the prefix the node represents. The current implementation assumes a 32-bit value (IPv4) though it can be extended to 128-bit value (IPv6).

4.2 Algorithm

```
node = T[0]; s = testdata[k]; node = table->trie[0];
pos = GETSKIP(node); branch = GETBRANCH(node);
adr = GETADR(node);
prefix=0; result=-1; /* stored in Register */
while(branch != 0) /* Not leaf node */
 {      node = table->trie[adr + EXTRACT(pos,branch,s)];
        if(pos)
                prefix<<pos;  /* skip 'pos' bits of the prefix */
         if (branch > n)
                prefix= prefix << (m+1)| branch;
              /* n=2^m-1, m is the number of bits
              representing the branch */
          if(GETSTRING(node)^prefix)
                break; /* Previous node contains largest prefix and
              interface stored in result */
          else
          {     pos = pos + branch + GETSKIP(node);
                branch = GETBRANCH(node);
                adr = GETADR(node);
                result = GETPORT(node);}
}
```

The search algorithm forms the bottleneck of the entire routing process and hence this needs to be designed very efficiently. Algorithm discussed above is used to search a string *s* in the routing table. EXTRACT (p,b,s) is used to search *s* in the routing table, where *b* is the number of bits starting at position *p*. Let the array representing the tree be T. The root is stored in T[0].

Each entry in Table 2 represents a node in the proposed approach, for routing table described in Table 1, with the corresponding branch, skip and pointer values. In addition to these three fields each node also has a 32-bit (IPv4) prefix represented by the node, which is not indicated in the table.

4.3 Working of the algorithm

The working of this algorithm is illustrated with an example. Consider the input string 101001. We start from root node number 0. We see that the branching factor is 3 and skip value is 0 and hence extract 1st 3 bits from the search string. These 3 bits represent a value of 5, which is added to the pointer, leading to position 6 in the

array. At this node the branching value is 2 and the skip value is 0 and hence we extract the next 2 bits. They have the value 0. However we check if the string (101) matches the prefix (101). Since it matches the search continues further. We now add the value of 0 to the pointer and arrive at position 13. At this node the branching factor is 1 and skip value is 0. They have a value of 1. We again compute to see if the string (10100) is same the prefix (10100). Since it matches we continue and add the value of 1 to 19 to obtain the pointer 20. Now see that this node represents a leaf node since the branching factor is 0. We now check to see if the string (101001) matches the prefix (101011). Since they don't match we use the previous value of the output port from the register to route the packet. The Figure 3 represents the path taken during the search.

Table 2. Array representation our approach

	branch	skip	pointer		branch	skip	pointer
0	3	0	1	11	0	0	0
1	1	0	9	12	0	0	0
2	0	2	0	13	1	0	19
3	0	0	0	14	1	0	20
4	1	0	11	15	0	0	0
5	0	0	0	16	0	0	0
6	2	0	13	17	0	0	0
7	0	0	12	18	0	0	0
8	1	4	17	19	0	0	0
9	0	0	0	20	0	0	0
10	0	0	0				

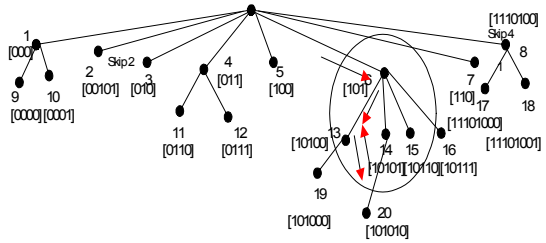

Figure 3. Trie traversal for the string 101001

Thus we see that this approach doesn't traverse the entire trie in case the string is not present. Also there is no separate storage for the prefixes as in case of LC-Trie. This approach checks for the match in the string at every step. However this is not computation intensive since the string to compare is already present in the cache and a *xor* on the bits could give us the result of comparison.

In the LC-trie approach after we traverse through the trie we perform a check for the string match in the base vector, which uses hashing technique, consuming at least one memory fetch. If there is a mismatch a check is done again on the prefix table and this requires hashing to check for a prefix match, which again requires another memory fetch. Thus compared to the LC-trie we save atleast two memory cycles for every routing lookup performed.

4.4. IPv6 Compatibility

The algorithm can easily be extended to IPv6 and to allow a maximum of 2^{37} entries. This ensures that the proposed data structure can support routing entries beyond 2005. This data structure also allows handling of a maximum of 1024 interfaces. The data structure for each node is described in the Figure 4.

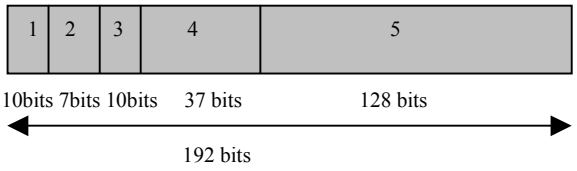

Figure 4. Proposed Data structure for node in routing table (IPv6)

5. Simulations & Experimental Setup

To test and verify our approach with the LC-Trie approach we have modified the test bed used by the authors of [1]. The features of this modified test bed are as follows:

1. It reads routing data from the routing table file, which is in a predefined format as discussed in the paper [1]. The routing file is an exhaustive list of routing entries (65536 entries for 16-bit pointer value).
2. The algorithm can be run over a number of times by specifying *n* as a command line argument. As the number of iterations increase, it gives a good estimate of the parameters under comparison.
3. Quick sort algorithm is used to sort the routing table entries.
4. We have also used two different approaches to compare the performance. One uses a function call to the search algorithm and the other is an inline function. However we have used the inline function results for our comparison.

In our implementation we have used routing tables similar to that provided by the Internet Performance Measurement and Analysis project

[7]. In order to compare the modified technique with LC-Trie approach the traffic was simulated and we used a random permutation of all possible entries in the routing table. The time measurements have been performed on sequences of lookup operations, where each lookup includes fetching the address from the array, performing the routing table lookup, accessing the nexthop table and assigning the result to a volatile variable. Some of the entries in the routing tables contain multiple nexthops. In such cases we select the first one listed as the nexthop address for the routing table, since we only consider one nexthop address per entry in the routing table. However for entries that didn't contain a nexthop address a special address that is different from the ones found in routing table was used.

The following equations were used in the computation of average and standard deviation of the samples (t_i).

Average Time (avg) = t_i/n

Std Deviation (std) = $(t_i^2 - n*avg*avg)^{1/2} /(n-1)$

Parameters analyzed

We have analyzed the effectiveness of our approach and compare our approach with the LC-Trie approach with respect to timing, storage and power consumption. The parameters considered in each of the cases are described below.

5.1. Timing

5.1.1 Building. Time taken to Build Routing table (B_t): This is the time taken for the algorithm to retrieve all the data from the Routing table file, sort them and build them with appropriate entries for future referencing.

Time taken to build nexthop table (N_t): This is the time taken to compute all the next hop addresses from the routing table data.

5.1.2 Sorting. Time taken to Sort the entries (S_t): Based on a seed value the routing table entries are stored in a temporary data structure in a random fashion, which is then sorted for building the routing table.

5.1.3 Searching.

Function Search: Time taken to search the string (using call to a function) based on *n* iterations.

F_{min}: Minimum time taken to search the string using function call.

F_{avg}: Average time taken to search the string using function call.

F_{std}: Standard deviation of the times for searching a string using function call.

F_{lps}: Average number of lookups/second using function call.

Inline Search: Time taken to search the string (using an inline function) based on *n* iterations.

I_{min}: Minimum time taken to search the string using Inline function call.

I_{avg}: Average time taken to search the string using Inline function call.

I_{std} : Standard deviation of the times for searching a string using Inline function call.

I_{lps}: Average number of lookups/second using inline function call.

5.2 Memory Utilization

B_m: Memory utilization in bytes for the base vector.

P_m: Memory utilization in bytes for prefix vector.

N_m: Memory utilization in bytes for nexthop vector.

Trie (T_m): Memory utilization for Trie.

6. Results

Following are the results for the comparison of LC-Trie approach and proposed approach with a fill factor of 0.5 (this is a good value based on the experimental results considering future updates) and a fixed branch at root (16). We have run this algorithm 100 times to get a good estimate of the values. This was run on an Intel Pentium II processor, 400Mhz and 256 MB RAM. The programs were written in C and complied with gcc compiler using optimization level –04.

From Table 3 we observe that time taken to build the trie is reduced by 0.14 seconds. This is mainly due to the fact that no additional computation is required to build the base and prefix vector. Also there is no overhead of building the nexthop table. The simulation results show that the proposed approach works 3.28 times better than LC-Trie approach when the prefix search is implemented as a function search and 4.11 times better when implemented as an inline function. Thus, for above mentioned system configuration we were able to achieve a lookup of approximately 6.6 Mpps in the average case. From Table 4 we observe that the proposed approach avoids the storage for base, prefix and nexthop vector and hence occupies 2.38 times lesser storage. Though the reduction in storage for the nexthop vector is not significantly high, the storage for the base and prefix vector is greatly reduced.

The processing power savings for the two approaches were compared using an implementation based on reconfigurable processor architecture from *Tensilica* (16/24 bit Xtensa ISA, 200Mhz, 0.18 um technology, 0.7 mm^2 core area, 0.8mW/MHz core power dissipation). The Routing table used in our power analysis is described in Table 1. The result

obtained is an estimate of power for one iteration. The results show that proposed approach and the LC-Trie approach consumes 4.615mW and 4.755mW for 20 lookups respectively (Table 5). This is a reduction of 0.14mW for 20 lookups. This is directly related to the fact that the time for lookup is less. The routing entries in our simulations are assumed to be stored in the DRAM and the storage power corresponding to that was computed for both the approaches. Since the storage requirements are reduced by factor of 2.38, it is expected that power consumptions will be less by that amount.

Table 3. Timing

Parameters	LC-Trie	Proposed	Savings(%)
B_t	0.57 sec	0.43 sec	24
N_t	0.05 sec	0 sec	100
S_t	0.37 sec	0.36 sec	2.7
F_{min}	5.01 sec	1.53 sec	69.4
F_{avg}	5.02 sec	1.53625 sec	69.4
F_{std}	0.01	0.0074402	-
F_{lps}	1308104	4283399	-
I_{min}	4.12 sec	1.0 sec	75.7
I_{avg}	4.12 sec	1.005 sec	75.7
I_{std}	0.01	0.0053452	-
I_{lps}	1590680	6553600	-

Table 4. Memory Utilization

Parameters	LC-Trie	Proposed
B_m	32769*16	0
P_m	32767*14	0
N_m	4*16	0
T_m	65537*4	65537*8

Table 5. Power consumed

Approach	No cycles	Proc. Config	Power/20lookup
LC Trie	5944854	0.8mW/Mhz	4.755 mW
Proposed	5769630	0.8mW/Mhz	4.615 mW

7. Conclusion & Future work

We have proposed a scalable and efficient algorithm for fast lookup based on modified LC-Trie technique. This algorithm performs about four times better in terms of access time in the average case as compared to the LC-Trie approach.

The proposed algorithm does approximately 6.6 million lookups per second on a 32 bit, 200Mhtz processor without considering caching of packets. Since the packet have certain amount of locality in them, caching could lead to better performance. The search algorithm forms the bottleneck in the lookup process. Hence computation kernels can be obtained and optimized by implementing at the hardware level. One such method is by creating tie instructions (in Xtensa) for the set of instructions that are executed more frequently.

8. References

[1] S. Nilsson and G. Karlsson, Fast Address Look-Up for Internet Routers, Proceedings of IEEE Broadband Communications 98, (April 1998), 9-18

[2] Venkatachary Srinivasan and George Varghese, Faster {IP} Lookups Using Controlled Prefix Expansion, Measurement and Modeling of Computer Systems, (1998), 1-10.

[3] Marcel Waldvogel, George Varghese and Jon Turner and Bernhard Plattner, Scalable High Speed {IP} Routing Lookups Proceedings of SIGCOMM '97, (1997), 25-36.

[4] Edward Fredkin. Trie memory. Communications of the ACM, (1960), 490-500.

[5] W Szpankowski, Patricia Tries again revisited, Journal Of the ACM '90, (Oct 1990 Vol.37, No. 4), 691-711.

[6] Tzi-Cker Chiueh and Prashant Pradhan, "Cache Memory Design for Network Processors", HPCA, (2000), 409-418.
http://ipdps.eece.unm.edu/2000/raw/18000884.pdf/

[7] "Intenet Performance Measurement Analysis Project", University of Michigan and Merit Network, [Online]: http://www.merit.

[8] Tzi-Cker Chiueh and Prashant Pradhan, High-Performance IP Routing Table Lookup Using CPU Caching, INFOCOM '99, (1999), 1421-1428.

[9] Pankaj Gupta "Algorithms for routing lookups and packet classification", A dissertation submitted to the department of computer science and the committee on graduate studies of Stanford university, Dec 2000,[Online]:
http://klamath.stanford.edu/~pankaj/thesis/thesis_2sided.pdf

[10] Tzi-cker Chiueh and Prashant Pradhan, Suez: A Cluster-Based Scalable Real-Time Packet Router, International Conference on Distributed Computing Systems, (2000), 136-143.

[11] P.Newman, G.Minshall, T.Lyon, and L.Huston, IP switching and gigabit routers. IEEE Communications Magazine, 35(1): (January 1997), 64-69.