Oriental Journal of Computer Science & Technology Vol. **4**(1), 51-58 (2011)

Advanced Algorithmic Approach for IP Lookup (IPv6)

PANKAJ GUPTA, DEEPAK JAIN, NIKHIL ANTHONY, PRANAV GUPTA, HARSH BHOJWANI and UMA NAGARAJ

Department of Computer Engineering, MAE, Pune - 412 105 (India).

(Received: April 30, 2011; Accepted: May 15, 2011)

ABSTRACT

Internet address lookup is a challenging problem because of increasing routing table sizes, increased traffic, higher speed links, and the migration to 128 bit IPv6 addresses. IP routing lookup requires computing the best matching prefix, for which standard solutions like hashing were believed to be inapplicable. The best existing solution we know of, BSD radix tries, scales badly as IP moves to 128 bit addresses. This paper presents a novel algorithm "Distributed memory organization" for lookup of 128 bit IPv6 addresses and "Asymmetric Linear search" on hash tables organized by prefix lengths. Our scheme scales very well when traffic on routers is unevenly distributed and in genera it requires only 3-4 lookups, independent of the address bits and table size. Thus it scales very well for IPv4 and IPv6 under such network conditions. Using the proposed techniques a router can achieve a much higher packet forwarding rate and throughput.

Key words: Best Matching Prefix, Longest Prefix Match, IP lookup, IPv6, Distributed Memory Organization, Mutating Binary search.

INTRODUCTION

The internet is becoming ubiquitous and has been exponentially and continuously growing in size. The number of users, networks and domains connected to the internet seem to be exploding. The 32-bit addresses of the IPv4 format are not sufficient to address the rapidly increasing users and domains, and will soon be exhausted. Hence the 128-bit IPv6 address format that has a much larger addressing capacity will soon replace the existing IPv4 address format. With exponential growth of the number of users and new applications (e.g. the web, video conferencing, remote imaging and multimedia), it is not surprised that the network traffic is doubling every few months. This increasing traffic demand requires three key factors to keep pace if the internet is to continue to provide good service: *link speeds*, *router data throughput*, and *packet forwarding rates*[1]. Readily available solutions exist for the first two factors: for example, fiber-optic cables can provide faster links, and switching technology can be used to move packets from the input interface of a router to the corresponding output interface at gigabit speeds. The packet forwarding process in a router involves finding the prefix in the routing table that provides the best match to the destination address of the packet to be routed. When a router receives a packet P from an input link interface, it must compute which of the prefix in its routing table has the *longest match* when compared to the destination address in the packet P. The result of the lookup provides an output link interface, to which packet P is forwarded. There is some additional bookkeeping, such as updating packet headers. But the major bottleneck of packet forward is IP lookup in the router table.

At present many lookup algorithms are available that produce high-speed lookups for the IPv4 addresses. But their performance degrades when they are scaled to provide lookup for the 128-bit IPv6 addresses. This performance degradation is due to increased number of memory access and

memory consumption as a result of the growth of the routing table size and address length in IPV6 [2-3].

In the present paper, we describe a novel lookup algorithm, *Distributed Memory Organization*, for lookup of 128-bit IPv6 addresses. The algorithm is capable of providing lookups for a maximum of 16 IPv6 addresses at a time. This is achieved by classifying the addresses stored in the routing table by analyzing the data of prefixes. Highly efficient lookup algorithm using asymmetric linear searching technique has been proposed. The storage mechanisms for these methods have also been optimized to significantly reduce the memory requirement and the average number of memory accesses. The rest of the paper is organized as follows. Section 2 describes the drawbacks to the existing approaches to IP lookup. In section 3, we propose IP lookups for IPv6, including memory organization mechanism, Asymmetric linear search algorithm. Section 4 gives some optimizations for searching technique and section 5 draws the conclusion and future works.

Existing approaches to ip lookup

In this section, we study some existing approaches to IP lookups and their problems. We discuss approaches based on trie-based schemes, range search method and hardware solutions, that can provide lookup for IPv6 addresses.

Trie Based Schemes

The most commonly available IP lookup implementation is found in the BSD kernel, and is a radix trie implementation [4]. If W is the length of an address, the worst-case time in the basic implementation can be shown to be O(W2). Current implementations have made a number of improvements on Sklowers original implementation. The worst case was improved to O(W) by requiring that the prefix be contiguous. Despite this, the implementation requires up to 32 or 128 costly memory accesses (for IPv4 or IPv6, respectively). Tries also can have large storage requirements[4]. Also, multibit tries improve lookup speed (for IPv4 addresses) with respect to binary tries, but only by a constant factor in the length dimension[5]. Hence, multibit tries scale badly to the longer IPv6 addresses.

Range search approach

The range search approach gets rid of the length dimension of prefixes and performs a search based on the endpoints delimiting disjoint basic intervals of addresses[5]. The number of basic intervals depends on the covering relationship between the prefix ranges, but in the worst case it is equal to 2N, where N is the number of prefixes in the routing table. Also, the best matching prefix (BMP) must be precomputed for each basic interval[11], and in the worst case an update needs to recompute the BMP of 2N basic intervals. The update complexity is O(2N). Since the range search scheme needs to store the endpoints, the memory requirement has complexity O(2N).

Hardware Solution

Hardware solutions can potentially use parallelism to gain lookup speed. For exact matches, this is done using Content Addressable Memories (CAMS) in which every memory location, in parallel, compares the input key value to the content of that memory location. Some CAMS allow a mask of bits that must be matched.

Although there are expensive so-called ternary CAMS available allowing a mask to be specified per word, the mask must typically be specified in advance. It has been shown that these CAMS can be used to do BMP lookups[6-7], but the solutions are usually expensive. Large CAMS are usually lower and much more expensive than ordinary memory. Typical CAMS are small, both in the number of bits per entry and the number of entries. Thus the CAM memory for large address/mask pairs (256 bits needed for IPv6) and a huge amount of prefixes appears (currently) to be very expensive. Another possibility is to use a number of CAMS doing parallel lookups for each prefix length. Again, this seems expensive. Probably the most fundamental problem with CAMS is that CAM designs have not historically kept pace with improvements in RAM memory. Thus a CAM based solution (or indeed any hardware solution) runs the risk of being made obsolete, in a few years, by software technology running on faster processors and memory. Recently Sangireddy et. al. suggests BDD based hardware address lookup engine, which can reduce the complexity of hardware by decreasing the actual effective nodes[8]. But it is still not scaled well to IPv6.

Proposed scheme for ipv6

Distributed Memory Organization

The prefixes stored in a routing table can be classified into several flows averagely depending on certain bits of them. For instance, we use bits 1, 2, 3 and4 (called ID bits) to classify the prefixes in the routing table into 16 categories as shown in Table 1. The key point is to store each category of the classified addresses in different memory modules so that high- speed lookups for a maximum of 16 IPv6 addresses is performed simultaneously

Table 1: Memory module allocation

Lookup Unit No.	Bits 1, 2, 3 and 4
1	0001
2	0010
3	0011
…	…
16	1111

For the prefix whose length is less than 4, we can expand it to the prefix with length of 4 by controlled prefix expansion technique in [9]. For example, the prefix 110* can be expanded as follows,

Table 1: The prefix expansion method

Before expansion	After expansion
110*	1101*
	1100*

According to our scheme the incoming IP address is classified into one of the 16 categories by the four ID bits. Then the search for the longest matching prefix for this incoming address starts in that memory module which contains the addresses of the same category. The algorithm for parallel lookup is given below

Pseudocode

Function

Lookup (Destination Address)
Use the ID bits of Destination Addresses to classify them.
Push the IP Addresses into the FIFO of the corresponding Lookup unit.
For each Lookup unit simultaneously **do**
While (FIFO not empty) **do**
Pop an address from the local FIFO.
Use binary lookup schemes to find BMP.
Push the Next Hop Address into Output cache.
End While
End Loop
End Function

The complete parallel lookup mechanism and the distributed memory organization is shown in Fig. 1 [8].

Asymmetric Linear Search

This is a very effective optimization over basic Linear Search in a situation where the traffic over the network is very unevenly distributed. A network is said to be unevenly distributed if packets frequently arrive on particular routers and rest of the routers in the network are rarely addressed.

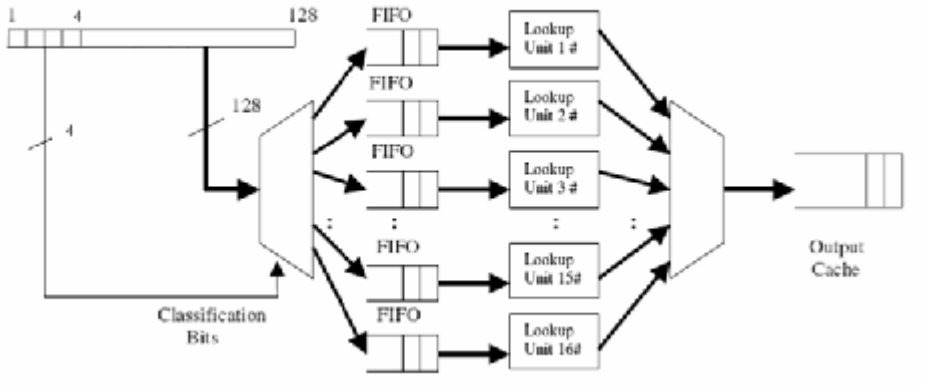
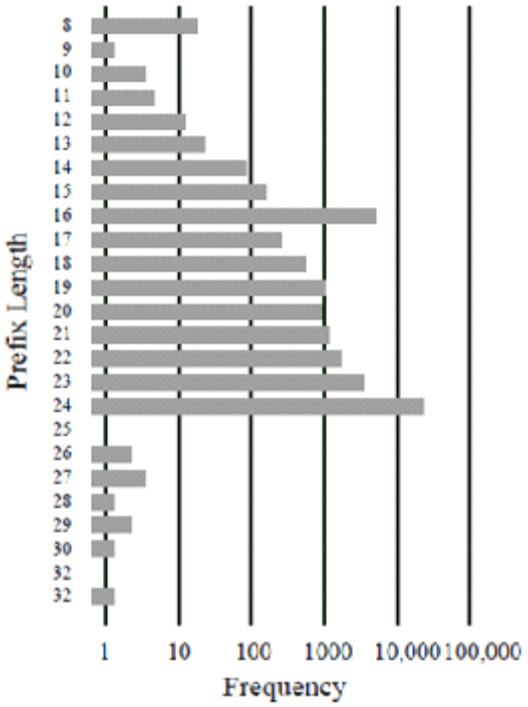


Fig. 1: Distributed memory organization and parallel lookup mechanism

Usually, the performance of general algorithms can be improved by tailoring them to the particular datasets they will be applied to. As can be seen in Figure 7, the distribution of a typical backbone router's forwarding table as obtained from [Mer96],



As say for 2,00,000 number of entries in lookup table, the probability for worst case behaviour may only be seen in 5.30% inputs. This technique is most beneficial when the network topology is such that the traffic is unevenly distributed in the network.

A counter is associated with each entry in the lookup table, each time an entry is matched, it moves one step upwards and its counter value increases. The storage for counter can be reduced by using proper storage data structure for lookup units. Proposed Storage Structure for lookup unit Is shown in Table 3. The IP addresses will always be in the sorted order based on their traffic to maintain asymmetricity, where count represents the number of packets arrived on that particular router.

Show some measures ,like a difference table b/w linear, asymmetric linear and binary in such conditions(uneven traffic distributiion).

the entries are not equally distributed over the different prefix lengths.

Asymmetric linear searching technique works in the corresponding lookup units. The incoming IP address will be fetched from the respective FIFO and then will be looked up in its related lookup unit, so while looking in lookup unit , our algorithm for searching will come into picture. The router entries will be arranged in decreasing order of their traffics. So in average case, exact matching prefix will be found in first few searches, only the worst case time will be equivalent to the no. of entries in the table. Although the probability for worst case time will be P(log N), where N represents the number of entries in lookup table. This probabilistic value for worst case assumes that the traffic is logarithmically unevenly distributed and it gives very good results.

Table 3: Storage structure for lookup units

IP Address	Count
123.237.70.212	146
123.237.70.200	143
…….	……
123.237.40.23	10

Then in next section show the optimization to mutating binary search and marker algorithm.(say "as marker algorithm has already been explained in (give refernece)")
Put stress on mutating binary.

I. Refinement to searching algorithm

A. Mutating Binary Search

We refine the basic binary search tree to change or *mutate* to more specialized binary trees each time we encounter a partial match in some hash table. We believe this a far more effective optimization than the use of basic binary search tree algorithm In the last section, we looked at prefix distributions sorted by prefix lengths and plotted a histogram in decreasing order of the traffic on each router. This resulting histogram led us to propose asymmetrical binary search, which can improve average speed.

Further information about prefix distributions can be extracted by dissecting the histogram: For each possible n bit prefix, we could draw 2n individual histograms with possibly fewer non- empty buckets, thus reducing the depth of the search tree.

When partitioning according to 16 bit prefixe5, and counting the number of distinct prefix lengths in the partitions, we discover a nice property of the routing data (Table 4). Though the whole histogram (Figure 2) shows 23 distinct prefix lengths with many buckets containing a significant number of entries, none of the "sliced" histograms contain more than 12 distinct prefixes; in fact, the vast majority only contain one prefix, which often happens to be in the 16 bit prefix length hash table itself. This suggests that if we start with 16 bits in the binary search and get a match, we need only do binary search on a set of lengths that is much smaller than the 16possible lengths we would have to search in naive binary search.

In general, every match in the binary search with some marker X, means that we need only search among the set of prefixes for which X is a prefix.

Table 3: Number of Distinct prefix Lengths in the 16 bit Partitions (Histogram)

Distinct Lengths	Frequency
1	4977
2	608
3	365
4	249
5	165
6	118
7	78
8	46
9	35
10	15
11	9
12	3

This is illustrated in Figure 3. On a match we need only search in the subtrie rooted at X (rather than search the entire lower half of the trie, which is what naive binary search would do.) Thus the whole idea in mutating binary search is as follows: *whenever we get a match and move to a new subtrie, we only need to do binary search on the levels of new subtrie.* In other words, the binary search *mutates* or changes the levels on which it searches dynamically (in a way that always reduces the levels to be searched), as it gets more and more match information.

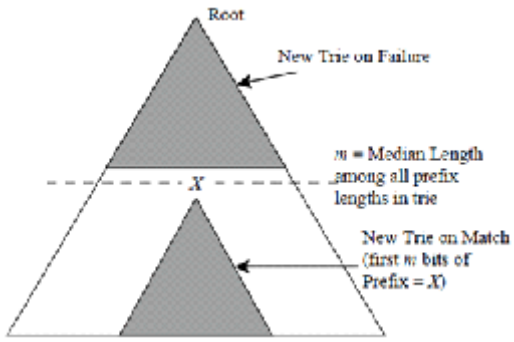


Fig. 3 : Showing how mutating binary search for prefix P dynamically changes the trie on which it will do binary search of hash tables

Thus each entry E in the search table could contain a description of a search tree specialized for all prefixes that start with E. This simple optimization cuts the average search time to below two steps (Table 4), assuming probability proportional to the covered address space. Also with other probability distributions,(i.e., according to actual measurements),we expect the average number of lookups to be around two.

As an example, consider binary search to be operating on a tree of levels starting with a root level, say 16. If we get a match which is a marker, we go "down" to the level pointed to by the down child of the current node; if we get a match which is a prefix and not a marker, we are done; finally, if we get no match, we go "up". In the basic scheme without mutation, we start with root level 16; if we get a marker match we go down to level 24, and go up to Level 8 if we get no match.

During basic binary search for an IPv4 address whose BMP has length 21 requires checking the prefix lengths 16 (hit), 24 (miss), 20 hit), 22 (miss), and finally 21. On each hit, we go

down, and on misses up. Using Mutating Binary Search, looking for an address see figure 4) is different. First, we explain some new conventions for reading Figure 4. we have multiple binary trees drawn on the left of the figure, labeled as Tree 1, Tree 2, etc. This is because the search process will move from tree to tree. Each binary tree has the root level (i.e., the first length to be searched) at the left; the upper child of each binary tree node is the length to be searched on failure, and whenever there is a match, the search switches to the more specific tree. Finally, Figure 4 has a number of prefixes and markers that are labeled as E,F,G,H,J for convenience. Every such entry in our example has E as a prefix. Thus rather than describe all the bits in E, we denote the bits E as…; the bits in say F are denoted as .... 111, which denotes the concatenation of the bits in E with the suffix 111. Finally, each hash table entry consists of the name of the node, followed by the bits representing the entry, followed by the label of the binary tree to follow if we get a match on this entry. The bmp values are not shown for brevity. Consider now a search for an address whose BMP is G in the database of Figure 4. The search starts with a generic tree, Tree 1, so length 16 is checked, finding E. among the prefixes starting with E, there are known to be only five distinct lengths (say 17, 18, 19, 20, 21, and 22). So E contains a description of the new tree, Tree 2, limiting the search appropriately. Using Tree 2, we find F, giving a new tree with only a single length, leading to G. The binary tree has mutated from the original tree of 32 lengths, to a secondary tree of 5 lengths, to a tertiary "tree" containing just a single length. Looking for J is similar. Using Tree 1, we find E. Switching to Tree 2, we find H, but after switching to Tree 4, we miss at length 21. Since a miss (no entry found) can't update a tree, we follow our current tree upwards to length 20, where we find J. In general, whenever we go down in the current tree, we can potentially move to a specialized binary tree because each match in the binary search is longer than any previous matches, and hence may contain more specialized information. Mutating binary trees arise naturally in our application (unlike classical binary search) because each level in the binary search has multiple entries stored in a hash table. as opposed to a single entry in classical binary search. Each of the multiple entries can point to a more specialized binary tree.

Table 4: Address (A) and Entry (E) Coverage for Mutating Binary Search

Steps	Usage		Balance	
	A	E	A%	E%
1	43.9%	14.2%	43.9%	14.2%
2	98.4%	65.5%	97.4%	73.5%
3	99.5%	84.9%	99.1%	93.5%
4	998%	93.6%	99.9%	99.5%
5	99.9%	97.8%	100.0%	100.0%
Average	1.6	2.4	1.6	2.2
Worst case	6	6	5	5

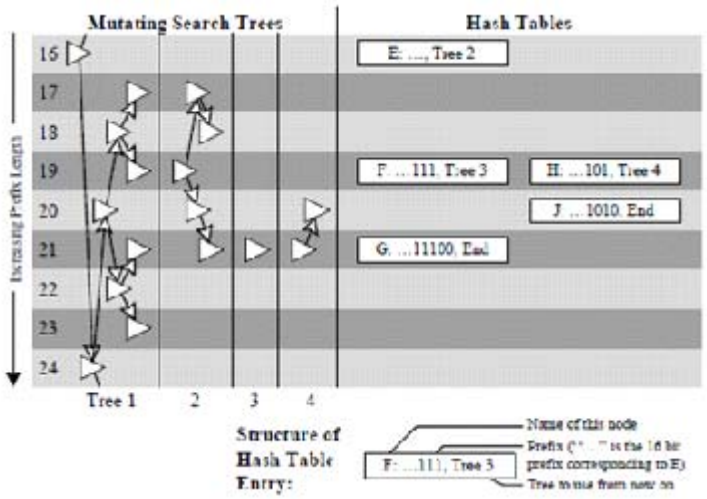


Fig. 4: Mutating binary search example

In other words, the search is no longer walking through a single binary search tree, but through a whole network of interconnected trees. Branching decisions are not only based on the current prefix length and whether or not a match is found, but also on what the best match so far is (which in turn is based on the address we're looking for.) Thus at each branching point, you not only select which way to branch, but also change to themost optimal tree. This additional information about optimal tree branches is derived by pre computation based on the distribution of prefixes in the current dataset. This gives us a faster search pattern than just searching on either prefix length or address alone.

Two possible disadvantages of mutating binary search immediately present themselves. First, precomputing optimal trees can increase the time to insert a new prefix. Second, the storage required to store an optimal binary tree for each prefix appears to be enormous. For now, we only observe that while routes to prefixes may frequently change in cost, the addition of a new prefix (which is the expensive case) should be much rarer. We proceed to deal with the space issue by compactly encoding the network of trees.

Conclusion and future works

We have designed a new algorithm for best matching search. The best matching prefix problem has been around for twenty years in theoretical computer science; to the best of our knowledge, the best theoretical algorithms are based on tries. While inefficient algorithms based on hashing [9] were known, we have discovered an extremely efficient algorithm that scales with the logarithm of the address size and so is very close to the theoretical limit of O(log logN) Our algorithm contains both intellectual and practical contributions. On the intellectual side, after the basic notion of binary searching on hash tables, we found that we had to add markers and use precomputation, to ensure logarithmic time in the worst-case.

Algorithms that only use binary search of hash tables are unlikely to provide logarithmic time in the worst case. Mutating binary trees is an aesthetically pleasing idea that 4 leverages off the extra structure inherent in our particular form of binary search.

On the practical side, we have a fast, scalable solution for IP lookups that can be implemented in either software or hardware. Our software projections for IPv4 are 80 ns and we expect 150– 200 ns for IPv6. Our average case speed projections are based on the structure of existing routing databases that we examined.

We expect most of the characteristics of this address structure to strengthen in the future, especially with the transition to IPv6. Even if our predictions, based on the little evidence available today, should prove to be wrong, the overall performance can easily be restricted to that of the basic algorithm which already performs well. With algorithms such as ours, we believe that there is no more reason for router throughputs to be limited by the speed of their lookup engine. We also do not believe that hardware lookup engines are required because our algorithm can be implemented in software and still perform well. For similar reasons, we do not believe that there is a compelling need for protocol changes to avoid lookups as proposed in Tag and IP Switching. Even if these protocol changes were accepted, fast lookup algorithms such as ours are likely to be needed at several places in the network. Future work on our algorithm includes theoretical work on a choice of balancing function, hopefully yielding an improvement over our ad-hoc heuristic functions. Other avenues of research include the choice of a heuristic function based on actual network traffic, and work on faster insertion algorithms. We are also trying to optimize the building and modification processes. Our algorithm belongs to a class of algorithms that speed up search at the expense of insertion; we are looking for other applications and generalizations of our algorithm.

In spite of potential improvements, we believe our algorithm is ready for practical use.

REFERENCES

1. M. Waldvogel, G. Varghese, J. Turner, and B. Plattner,"Scalable high speed IP routing lookups", in Proc.SIGCOMM'97, Cannes, France.
2. S. Deering and R. Hinden., "Internet Protocol,Version 6 (IPv6)", Specification RFC 1883, IETF. [Online] (1995).
3. C. Huitema, "IPv6: The New Internet Protocol", Englewood Cliffs, NJ: Prentice-Hall (1996).
4. H. J. Chao, "Next Generation Routers", Proceedings of the IEEE, 90(9): (2002).
5. M. A. Ruiz-Sanchez, E. W. Biersack, W. Dabbous, "Survey and Taxanomy of IP Address Lookup Algorithms", *IEEE Network,* **15**(2): 8-23 (2001).
6. A. McAuley and P. Francis, "Fast routing table lookup US- ing CAMS", In Proceedings of INFOCOM, pages 1382-1391 (1993).
7. A. J. McAuley, P. F. Tsuchiya, and D. V. Wilson, "Fast multilevel hierarchical routing table using content addressable memory", U.S. Patent serial number 034444, Assignee Bell Communications research Inc Livingston NJ, (1995).
8. K. Venkatesh, S. Aravind, R. Ganapath and T. Srinivasan,"A High Performance Parallel IP Lookup Technique Using Distributed Memory Organization", In Proc.ITCC'04.
9. Keith Sklower. A tree-based routing table for Berkeley Unix. Technical report, University of California, Berkeley, (1993).