IP Lookup in IPv6 Networks

Shuping Liu
Networking Technology Lab.
Helsinki University of Technology, Finland
Email: sliu1@cc.hut.fi

***Abstract*: The IP Lookup Process is a key bottleneck in routing because of increasing routing table sizes, increasing traffic, higher speed links, and migration to 128-bit IPv6 addresses. The IP routing lookup involves computation of the best matching prefix (BMP) for which existing solutions, such as BSD Radix Tries, scale poorly when traffic in the router increases or when employed for IPv6 address lookups. In this paper a distributed memory organization technique for the routing table was described, which performs extremely well for IPv6 address lookup. This mechanism provides lookup for a maximum of 16 IPv6 addresses simultaneously. In each lookup subsystem, a scheme for BMP using binary search on hash tables organized by prefix length was proposed. An optimized storage mechanism for binary search on hash table scheme is also presented. Using the proposed techniques a router can achieve a much higher packet forwarding rate and throughput.**

Keywords— IP lookup, IPv6, Distributed Memory Organization, Binary search, Hash table.

I. 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 algorithms using optimized binary searching techniques on hash tables have 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, binary search on hash table and marker storage algorithm. Section 4 gives the performance analysis and simulation result for the proposed IP lookups. Session 5 draws the conclusion.

II. 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.

A. *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 $O(W^2)$. 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.

B. *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).

C. *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 slower 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.

III. PROPOSED SCHEMES FOR IPV6

A. *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.

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

Table 1 Memory module allocation

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,

Before expansion	After expansion
110*	1101* 1100*

Table 2 the prefix expansion method

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.

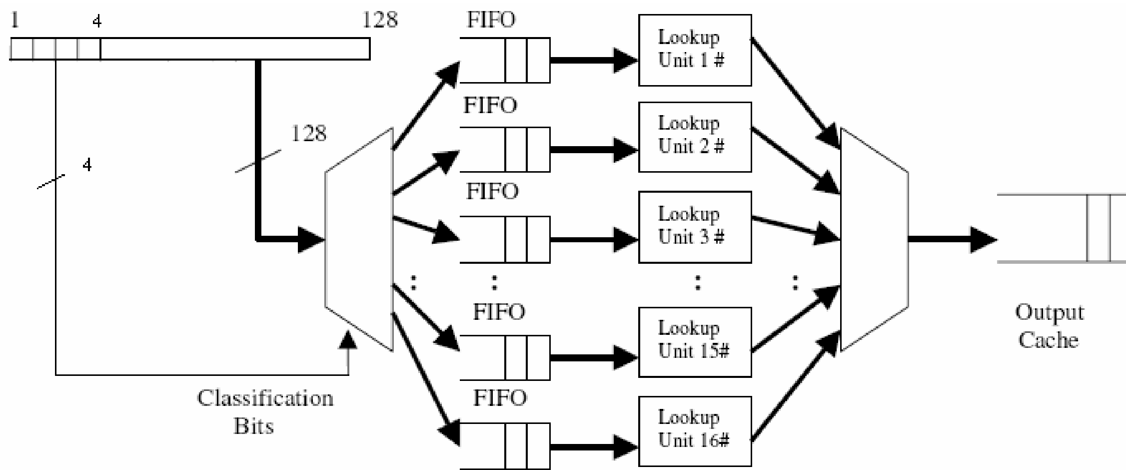

Figure 1 Distributed memory organization and parallel lookup mechanism

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 Figure.1 [10].

B. Binary Search on Hash Table

In each lookup subsystem, the prefixes are organized as hash table by prefix length, see Figure 2. The point of scheme is to do binary search on the hash table organized by the prefix length [1].

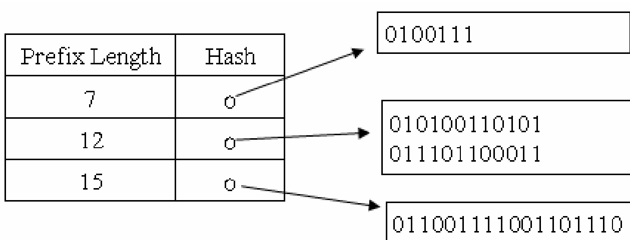

Figure 2. Hash tables for each possible prefix length

Binary search is to search a sorted array L by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty. The algorithm for basic binary search approach is given below.

Function BasicBinarysearch (A)
Initialize the search range S as the whole array L
do
Let i correspond to the middle level in S
Extract the first L[i].length bits of A to A'
Search (A', L[i].hash) // search for A'
If found **then** set S= lower half of S
Else set S= upper half of S
While S is not a single entry
End Function

The search time of binary search in the worst case is $\log_2 N$, where N is the length of array. On the other hand, the complexity of linear search is O(N). It is obvious that binary search strategy is better than linear search strategy. In figure 2, N is the number of prefix lengths (3), and the start point is the middle hash table whose prefix length is 12.

However, for binary search to work, we need *markers* in tables to direct binary search to look for matching prefixes of greater length. Here is an example to illustrate the need of markers.

For instance, we have prefixes P1=0, P2=00, P3=111. Suppose the prefix we search for is 111, the search starts from the prefix P2. Since the P2 does not match, the search

algorithm should proceed further till a match is found. But since there is no indication of the path to be taken, this approach fails to arrive at the best matching prefix.

To solve this problem, the binary search on hash table using markers proposed in [1] performs fairly well with respect to the number of memory access needed for obtaining a best matching prefix for IPv6 destination packet address. Cooperating with this search algorithm, our distributed memory organization technique provides multiple lookups keeping the number of access for each lookup to be logarithmic. We store markers, which are the first n bits of the prefix to be inserted, where n is the length of the prefix at that level. These markers are stored at the levels, which would be reached while searching for a matching prefix, with prefix length shorter than the prefix being inserted. So in the above example, we add a marker entry 11, to indicate P3, at the level containing P2 to direct the binary search to the lower half of the routing table for a better match.

With markers in each hash table, we start binary search on hash table corresponding to the median length of array L. If we find match, we search the upper half of L; if we fail we search the lower half of L. The procedure is repeated until we find BMP for the destination address. The procedure with marker is shown in Figure 3.

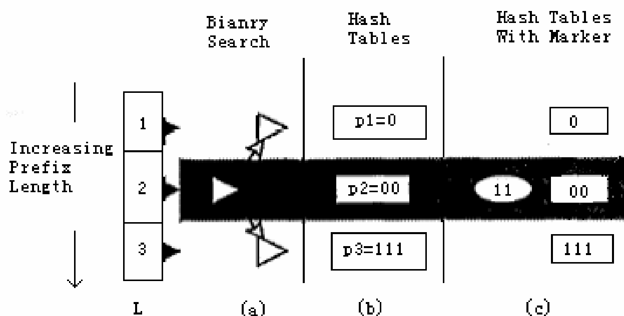

Figure 3 Binary search on Hash tables

Also the number of markers to be stored for each prefix is an important issue to be handled. Inefficient usage of markers will degrade the performance of the binary search and also increase the memory consumption. So an efficient approach to store markers is proposed in the next section. This approach enables efficient storage of markers when compared to the marker requirement stated in [1].

C. Marker Storage Algorithm

Our approach to store markers for a prefix is based on the bit pattern of the prefix. As already explained it suffices to store markers in those levels that would be visited by the binary search and whose length is shorter than that of the prefix to be inserted. The algorithm for our marker storage is given below.

```
Function MarkerStore (prefix)
    Initialize count = 0 , Level = 0
    Initialize bin = 0000
// Scans the prefix to find the length of prefix
    count = Length(prefix)
// Find the binary of the length 'count'
    bin = Binary (prefix)
// Scan this 'bin' for number of 1's
    count = Scan (bin)
    For I = count step -1 loop till I > 0 do
    Level = Level + (2 ^ I)
    Add a marker entry for the prefix in
      the level indicated by "Level"
    Search for BMP of marker and store
      it in the BMP field of the marker.
    Next I
End Function
```

For instance, if the prefix is P1=11001, then it should be inserted in the level 5 (0101 in binary). The number of 1s in the binary format of 5 is 2. Based on the above algorithm, the Level would become 4, which is the only level that would be reached during the search for a prefix of length 5 and whose length is smaller than 5. Hence a marker is added to level 4 for the prefix whose length is 5.

This marker storage algorithm is efficient as the number of potential parents for storing the markers is optimized in comparison to the existing approach stated in [1]. Also, since the number of markers stored for the prefix to be inserted is reduced, the overhead of marker insertion during the prefix insertion process is also reduced.

Consider the prefixes P1=1, P2=00, P3= 111. Now according to the marker storage logic explained above, marker for P3 will be stored at the level containing P2. Now when a prefix 110 is to be searched, the search starts at P2 and proceeds to the lower half of the table since a matching marker is available. But the best prefix match is available in the upper half of the hash table. Such a marker misleads the searching algorithm. A solution for this misleading marker problem has been proposed in [1]. A new field called BMP is stored for each marker. This field contains the best matching prefix of that marker. When we use the misleading marker and fail to arrive at the best matching prefix, the value in the BMP field of the latest marker arrived at is the longest matching prefix for the destination address.

IV. PERFORMANCE MEASUREMENT

A. Analysis of the Queuing System

We employ queuing theory to model the lookup subsystem, which is modeled as a pure waiting system. We assume that the arrival process of the incoming IP addresses is Poissonian process with mean arrival rate as λ. The service process for each lookup unit is considered to IDD and exponentially distributed with the mean service rate μ. Here in our lookup algorithm, the number of service stations S (i.e. number of memory banks) is 16. Hence the arrival rate for each queue is λ/16. The input destination IP address can be serviced in any of these stations based on the bit patterns as explained in the section 3. Now the service time for each lookup unit is $T_s = 1/\mu$. Since we have S lookup units, the arrival rate for each unit reduces, while the service rate is still μ. The state transition diagram for each lookup unit is as in Figure 4.

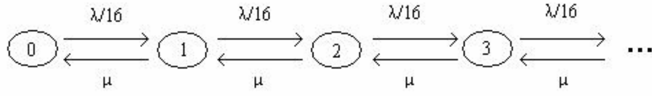

Figure 4 The state transition diagram for each lookup unit

For the whole lookup subsystem, 16 lookup units can be modeled as one server with the mean service rate as 16μ while the arrival rate for the whole system is λ. The corresponding state diagram is shown in Figure 5.

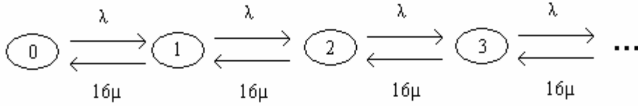

Figure 5 The state transition diagram for the whole lookup subsystem

Local balance equations give that,

$$\pi_{i-1} * \lambda = \pi_i * 16\mu \quad (1)$$

with the help of $\sum_{i=0}^{\infty} \pi_i = 1$, we get

$$\pi_i = \left(\frac{\lambda}{16\mu}\right)^i \pi_0 \quad (2)$$

Where $\pi_0 = \frac{16\mu - \lambda}{16\mu}$.

Thus the following performance measures can be obtained,

Utilization factor:

$$\rho = \frac{\lambda}{16\mu} \quad (3)$$

Mean queue length for memory module:

$$x = \frac{\rho}{1-\rho} = \frac{\lambda}{16\mu - \lambda} \quad (4)$$

Average delay time for an incoming address:

$$D = \frac{1}{16\mu} * \frac{1}{1-\rho} = \frac{1}{16\mu - \lambda} \quad (5)$$

Average waiting time in queue for an incoming address:

$$w = \frac{1}{16\mu} * \frac{\rho}{1-\rho} = \frac{1}{\mu} * \frac{\lambda}{16\mu - \lambda} \quad (6)$$

Now in our model we use 16 memory modules to store the route prefixes. Let us assume that λ/μ = 4/1. Now ρ = 1/4, using equation (3). Other parameters that are calculated based on this, using equations (4), (5) and (6), are

$$x = \frac{1}{3}$$

$$D = \frac{4}{3} * \frac{1}{\mu}$$

$$W = \frac{1}{3} * \frac{1}{\mu}$$

Now it is necessary to prove that the queue size is not large when there is a burst in traffic. Also we need to show that the waiting time is less for every IP Packet in the queue. As the network payload cannot be more than 0.8 (which is commonly adapted for by networking applications), the delay T_s will be less of the order of 0.1. As a result, the average queue size found using (4) with the above parameters is 4, which is within the very safe range. In this limit case, the total delay time is 0.031 seconds, and the waiting time is 0.025 seconds.

In this example, the number of lookup is increased by a factor of 16 when compared to the conventional method. Hence, small queues suffice each lookup unit to implement the proposed algorithm while number of lookups is increased.

B. *Simulation results of Binary on Hash Table*

Simulations were conducted to determine the number of memory access for both the binary search on hash table algorithms. The binary search on hash table mechanism requires a maximum of 7 memory accesses to find the longest matching prefix for a 128-bit IPv6 address.

Simulation results of the algorithms described above were used to construct the graphs shown in Figure. 6. It was observed that the average number of memory accesses for the lookup of a 128-bit IPv6 address is 6 for optimized binary search scheme.

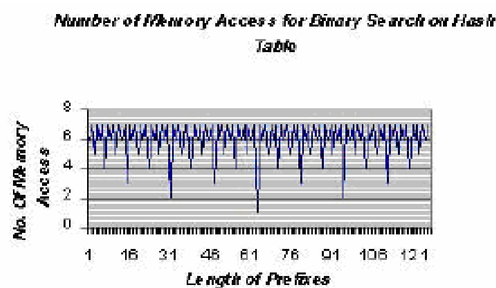

Figure 6 Simulation Result

V. CONCLUSION

We have proposed an algorithm for the best matching prefix search for the next generation IPv6 addresses that uses the hash based approach involving optimized storage of markers. This approach is extremely efficient that scales with the logarithm of address size and so is very close to the theoretical unit of O (logN).

By classifying the prefixes in the routing table, we are able to provide a maximum of 16 simultaneous lookups. This drastically increases the packet-forwarding rate of a router even for IPv6 address formats. We have also optimized the marker storage methodology for searching the hash table so that the memory requirement for the markers reduces considerably.

We have also analyzed the system performance based on queue theory for the proposed algorithms. The simulation were also been done whose results were used to construct a graph between the number of memory access and the length of the prefix to be searched. From these results it is observed that the proposed methods perform better than the existing algorithms for the lookup of IPv6 addresses.

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. (1995), “Internet Protocol, Version 6 (IPv6)”, Specification RFC 1883, IETF. [Online]. Available HTTP: http://www.ietf.org/rfc/.

[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, Vol.90, No.9, September 2002.

[5] M. A. Ruiz-Sanchez, E. W. Biersack, W. Dabbous, “Survey and Taxanomy of IP Address Lookup Algorithms”, IEEE Network, Vol.15, Issue 2, pp.8-23, March-April 2001.

[6] A. McAuley and P. Francis, “Fast routing table lookup US- ing CAMS”, In *Proceedings of INFOCOM,* pages 1382-1391, March-April 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, January 1995.

[8] R. Sangireddy and A. K. Somani, “High-Speed IP Routing With Binary Decision Diagrams Based Hardware Address Lookup Engine”, IEEE on Selected Areas in Communications, Vol. 21, No. 4, May 2003.

[9] V. Srinivasan, G. Varghese, “Fast IP Lookups Using Controlled Prefix Expansion”, ACM TOCS, Vol.17, pp.1-40, Feb.1999.

[10] K. Venkatesh, S. Aravind, R. Ganapath and T. Srinivasan, “A High Performance Parallel IP Lookup Technique Using Distributed Memory Organization”, In Proc. ITCC’04.

[11] B. Lampson, V. Srinivasan, G. Varghese, “IP Lookups Using Multiway and Multicolumn Search”, IEEE/ACM Transactions on Networking, Vol.7, No.3, June 1999.