IP Address Lookup Using A Dynamic Hash Function

Xiaojun Nie
Carleton University
xnie@math.carleton.ca

David J. Wilson
Alcatel
david.j.wilson@alcatel.com

Jerome Cornet
Alcatel
jerome.cornet@alcatel.com

Gerard Damm
Alcatel
gerard.damm@alcatel.com

Yiqiang Zhao
Carleton University
zhao@math.carleton.ca

ABSTRACT

The explosive growth of the Internet and of new applications over IP has made Internet routers the bottleneck in enabling higher speed communications. One of the more resource intensive functions of a router is the IP address lookup. This paper proposes a new IP address lookup algorithm that improves the performance and memory requirements of a hash-based lookup by exploiting the statistical repartition of prefixes in the forwarding table. Prototyping has shown that only one main memory access and several fast cache memory accesses are needed to perform a lookup on average. Furthermore, the configuration of the data structures can be tuned to control both memory usage and lookup performance.

1. Introduction

The basic function of Internet routers is to forward packets between networks based on network layer information and routing tables. In recent years the IP address lookup has become more difficult since the introduction of Classless Inter-domain Routing (CIDR) [1], in which a full longest prefix match (LPM) lookup is required to find the next hop.

As routers handle faster line rates, and as the Internet grows, more traditional LPM algorithms, such as the binary tree [2], Patricia tree [3], and path-compressed tree [4], have difficulty maintaining good performance. To alleviate this bottleneck many new LPM algorithms have been proposed to increase the lookup rates. Although most of these new algorithms can perform lookups at gigabit line rates, they typically suffer from one of many problems, such as complicated updates, poor worst case lookup times, or lack of scalability.

In response to the pitfalls listed above, we propose a novel scheme intended to speed up the search rate without sacrificing update performance, the worst case search time or scalability.

The proposed algorithm exploits the statistical repartition of the prefixes, to speed up a hash-based lookup, by controlling the hash function to distribute the hash collisions more evenly across the entire forwarding table. The result is an algorithm with configurable worst case performance, a simple updating algorithm, and easy implementation in different environments.

The remainder of this paper is organized as follows: Section 2 provides an overview of some of the related work in this field. The proposed algorithm will be described in detail, including the data structures, the search operation and update operation, in Section 3. Section 4 analyzes the performance of the algorithm based on the results of prototyping. Finally, conclusions of this work are given in section 5.

2. Related work

Three earlier and simpler IP lookup schemes, binary tree, Patricia tree and path-compressed tree were introduced, to perform the LPM, in the early stages of CIDR, but they currently do not satisfy the requirements of packet forwarding with the rapid development of the Internet.

To solve this problem, many new algorithms have been proposed. In general, these new algorithms can be categorized into software-based algorithms [5][6][7], which improve their search performance mainly through efficient data structures, and hardware based algorithms [8][9][10][11][12][13] that achieve an improvement in the IP lookup by maintaining a subset of the routing table in faster cache memory. There are other proposed algorithms [14][15][16][17][18], some of which are neither based on software nor hardware, but on the protocol. All of these algorithms have their own advantages based on the implementation environment, the routing table, flexibility, scalability, or even the router [19][20][21].

However, when these lookup algorithms employ complex data structures or technologies to speed up the lookup process, it is often at the cost of other performance measures like update time or worst case performance.

Some existing lookup schemes only concentrate on the fast IP address lookups at the expense of table updates. In the Lulea algorithm [23], updating a prefix entry will change all the entries in the subtree rooted at the updated node, which in the worst case is the entire forwarding table. In the DIR-24-8 algorithm [9], updating a prefix may involve 2^{16} prefix entries being changed in the worst case. Although route updates do not happen as frequently as address lookups, a slow update rate might have a negative impact on the address lookups.

To ensure fast lookup speed on average, some lookup schemes ignore the lookup rate in the worst case for the reason that the worst case is statistically improbable. In the multi-way and multi-column search algorithm [22], five main memory accesses and 44 cache memory accesses in the worst case are required for the routing database Mae-East. And the probability of hitting the worst is more than 1 percent or higher if the length distribution is considered, although only two main memory accesses are needed on the average. However, the worst case might happen frequently in a short time, resulting in a decline in router performance.

Scalability is an important performance metric in efficient lookup schemes, but it is too often ignored by many researchers. The Lulea algorithm [23] requires only 150 to 160K bytes for a database with 40,000 entries, but requires cache memory to hold the lookup table to achieve desirable performance, whereas in fact, not all routers can provide enough cache memory to hold this table. With highly scaleable algorithms, the high-speed performance will not change considerably for different routing databases or different router characteristics.

3. Proposed Algorithm

In this section, we propose a new algorithm based on a hash table (for fast lookups), in which the hash function is derived from the statistical repartition of the target prefixes.

3.1 Basic Idea

The proposed algorithm is based on the idea of the structure of a book, in which specific content can be found quickly using the book's table of contents. However, unlike a real book the contents would be grouped together into chunks of similar and small size (like a page), rather than by topic. The "dense topics" would have a detailed table of contents, and the "sparse topics" would have a broad one.

In our algorithm, the prefixes (contents of the book) are grouped together through the use of masking. The mask length is selected so that enough entries, but not too many, share the same root prefix. They will be packaged together in a "small search group", which can be found through the use of a hash table.

The prefix length to use while searching is stored in a small table called the "index table", and is calculated based on the number of entries that share the same prefix.

The hash table lookup is composed of two operations: masking off the target address with a mask whose length is specified in the index table, and then performing a regular hash lookup.

This combination can be viewed as a way of dynamically changing the hash function based on the repartition of the entries in the forwarding table.

On the implementation side, to speed up the search rate, the index table will be placed into an on-chip memory such as cache memory, and the small search group will be prefetched into the cache memory before being searched.

Figure 1 shows the basic process of an IP lookup using the index table. Given an incoming IP address, we first check the index table to obtain the group (or hash key) that the address belongs to. With the hash key and the hash function, we are able to find the small search group in the hash table in one step. To find the best matching prefix (BMP) for the address, we only need to search the small search group. Consequently, we reduce the search range size from the whole database to the small search group.

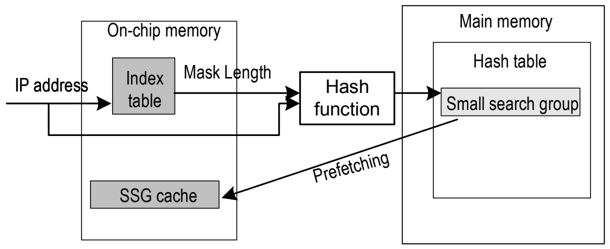

Figure 1: Basic process of the IP lookup

3.2 Data Structures

The data structures of the proposed algorithm consist of three parts: an index table, a hash table and small search groups, which will be individually described as follows.

3.2.1 Index Table

The index table is an array of 256 elements, each of which has two fields: A head and a sub index table. The data structure of the index table is shown in Figure 2. The head has four possible values: 0, 1, 2 or 3, each of which explained as follows:

'0': There is no matching prefix in the routing table for the input address.

'1': The first 8 bits of the input IP address are used as the hash key to perform a hash table lookup.

'2': The first 16 bits of the input IP address are used as the hash key to perform a hash table lookup.

'3': The first 16 or 24 bits of the input IP address are used as the hash key. In this case, the second 8 bits of the address are used to check the sub index table. If this 8-bit sequence is found in the sub-index table, the first 24 bits of the address are used as the hash key. Otherwise the first 16 bits of the address are extracted as the hash key.

Each index table element contains an 8-bit head and 256-bit sub index table representing all of the 256 possible values for the second 8 bits of the address. Each entry requires 9 bytes of memory memory. So the size of the index table will be 2.25K bytes, which is small enough to be placed into a cache memory.

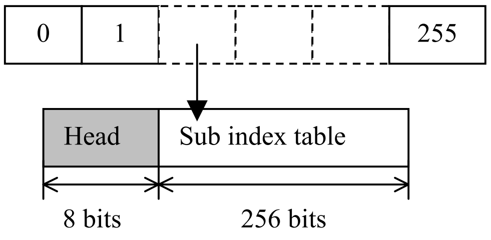

Figure 2: Data structure of the index table

3.2.2 Hash Table

The hash table in this algorithm is an array of *n* elements. The structure of a hash table element is shown in Figure 3. The hash table element has four fields: Identifier, SSG size, SSG pointer and collision pointer, which are explained below.

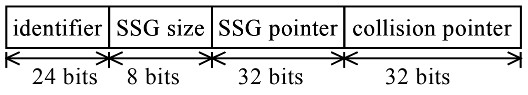

Figure 3: Data structure of hash table element

Identifier: Differentiates hash keys mapping into the same hash index from one another. This field width is 24 bits. If a hash key is less than 24 bits, then zeros are added to its left.

SSG Size: The size of the small search group pointed to by the SSG pointer.

SSG Pointer: The addresses of a small search group in which the BMP can be found, if it exists.

Collision pointer: The address of the next link in a linked list structure deployed to solve the hash collisions.

According to the structure in Figure 3, each hash table element is 12 bytes long, therefore, the hash table will require at least 12*n* bytes of memory.

3.2.3 Small Search Groups

A small search group is an array of variable size ranging from 1 to 16 elements. The elements of a small search group are sorted in descending order of prefix length. Each small search group element consists of two 32-bit long words. The first word represents part of a prefix (24 bits), an extension flag (1 bit) and the prefix length (7 bits). The second word is the pointer to the next hop. The structure of a small search group is shown in Figure 4.

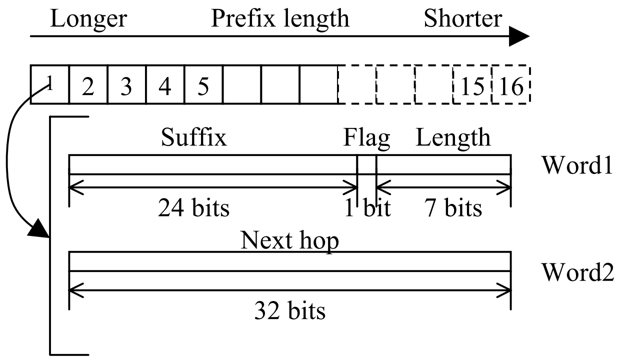

Figure 4: Data structure of a small search group

For a prefix of length *l*, and if its first *m* bits are its hash key, then the remaining *l-m* bits (the suffix) are put into the small search group. If the suffix is less than 24 bits long, it is extended to 24 bits by trailing zeros. If *l* equals *m*, the suffix is empty. In this case, 24 zeros are put in this field. If *l* is less than *m*, the prefix has to be extended to $2^{(m-l)}$ prefixes with length *m*, which are called "extended prefixes". If a prefix is an extended prefix, we use a 1-bit field called "extension flag" that is set to one to represent this. The prefix length field has 7 bits, which tells which part of an IP address is used to compare the suffix.

In a small search group, all prefixes are stored in order of decreasing length for linear search. That means that the search starts the prefix with the longest prefix length. When a match occurs, the search is terminated without going through all prefixes to find the BMP.

Since each element in the small search group has a size of 8 bytes, the maxim size of a small search group is 128 bytes, which can be fit into four 32-byte cache lines.

3.3 Lookup operation

Given an input address, finding the BMP from the lookup table can be described using the pseudo code of Figure 5.

The first 8 bits of the address are extracted as the index to the index table to get the head. If the head is '0', the search is terminated and the default next hop is returned. If the head is '1' or '2', the first 8 bits or 16 bits of the address, respectively, are used as its hash key to do hash lookup. If the head is '3', the second 8 bits of the address are taken to check the sub-index table. If the value is found in the sub-index table, the first 24 bits of the address are extracted as the hash key. Otherwise, the hash key is the first 16 bits of the address

```
Search ($addr$)
 $R_{index}$ = Checking $Index\ table$;
 if $R_{index}.head$ is zero, then search is done.
 else if $R_{index}.head$ is '1', then $key$ = the first 8 bits of $addr$
 else if $R_{index}.head$ is three, then checking sub index table
        if found, then $key$ = the first 24 bits of $addr$
 else $key$ = the first 16 bits of $addr$
 $R_{hash}$ = Search the hash table with $key$
 If $R_{hash}.size$ equals to one, then checking this prefix
 else prefetch small search group pointed by $R_{hash}.pointer$ and search the small search group in linear.
```

Figure 5. Pseudo code of the IP lookup

The result of the hash lookup is a pointer to the next hop if the size of the small search group is one or a pointer to a small search group if the size is greater than one. In the latter case, the small search group will be prefetched into cache memory to do a fast linear search.

3.4 Updates

There are three kinds of updates in the routing table: change of next hop, insertion of a prefix, and deletion of a prefix. Among them, the change of the next hop is the simplest update.

The approach of changing the next hop is similar to searching for an IP address in the routing table, which requires one lookup in the hash table, and checking the index table and small search group. Once the address has been located the next hop pointer simply needs to be updated.

The update of a new prefix may in some cases change the index table, the head or sub index table, which involves one or two fast memory accesses. Usually the size field of a hash table element will be changed during the updates; other fields will be modified when a new hash table element is created. In the latter case, there is one prefix change involved for small search group since the size of the small search group is one for new hash table element. Updating a prefix in the small search group is the most time consuming operation, as inserting or deleting the prefix will change the prefix length order in this small search group; this will cause the other prefixes in this group to be reorganized. But the modifications only happen in this small search group, which has a limited size of 16. Therefore, in addition to one main memory access to the hash table, updating a prefix will involve 16 main memory modifications in the worst case.

4. Performance Analysis

The proposed algorithm has been implemented on a general-purpose processor with 256K Bytes of cache memory and 64M bytes of main memory. The implementation has been done in C, and uses four snapshot databases from the Internet Performance Measurement and Analysis (IPMA) project [24], which include Mae-East, Mae-west, AADS and PAIX. Each database has a different size, as shown in Table 1.

Table 1: Size of the test routing databases

Database	Mae-East	Mae-west	AADS	PAIX
Size	24446	29601	29349	10859

In this section, performance metrics such as the number of memory accesses, the memory usage of the hash table and the update rate will be analyzed according to implementation results.

4.1 Memory accesses

In the implementation, two kinds of memory accesses are involved: main memory access and cache memory access, both of which are individually analyzed.

4.1.1 Main Memory Accesses

The hash lookup rate represented by main memory accesses is decided by hash collisions. So the average hash collisions reflect the average main memory accesses, while the maximum hash collisions indicate the worst-case main memory accesses.

Figure 6 shows the relation between the average hash collisions and the hash table size, while the changes of maximum hash collisions against the hash table size is given in Figure 7.

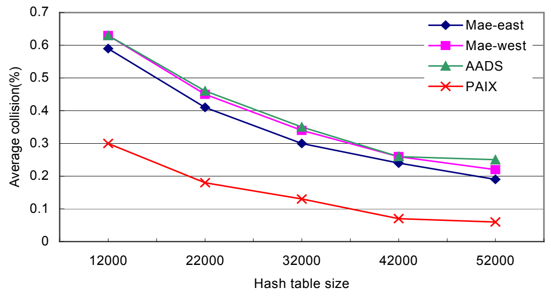

Figure 6: Average hash collisions vs. hash table Size

In the proposed algorithm, we expect that both the average and maximum number of hash collisions are minimal. Since the number of hash collisions is related to the table size, we will analyze the main memory accesses based on the hash table size. From Figure 6, we can see that the average number of hashes collisions drops when we increase the hash table size. It also implies that with a fixed hash table size, the number of hash collisions are proportional to the size of the routing database..

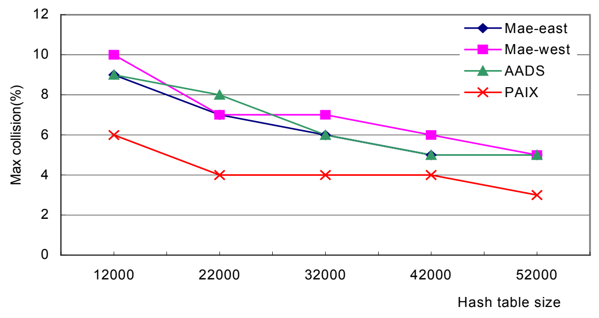

Figure7: Maximum hash collisions vs. hash table size.

Figure7 indicates that a large hash table size can improve the worst case, but the improvement is significant only when the hash table size is very large. Therefore, the hash table size should be as large as possible. With the hash table size of 52000, the average number of main memory access is between 1.07 and 1.35, while the number of worst-case main memory accesses ranges from four to six.

4.1.2 Cache Memory Accesses

Checking both the index table and searching small search groups involve memory access to the cache. The results of checking the index table are given in Table 2, while Table 3 lists the results of searching small search group.

Table 2: Number of cache memory accesses in the index table.

# of accesses	Mae-East	Mae-West	PAIX	AADS
1	48.24 %	55.40%	65.32%	54.02%
2	51.76 %	44.60%	34.68%	45.98%
Averag	1.52	1.45	1.35	1.46

The results in Table 2 show that the average number of cache memory accesses is around 1.5, and the worst case number is 2 for all the four databases.

From the result of Table 3, we can see that more than 80 percent small search groups had a size of one. That means that more than 80 percent small search groups do not need to be prefetched into cache memory. So the average number of cache memory accesses is only around 0.5, although the number reaches 16 in the worst case.

Combined with the previous results, to search an input address, less than three cache memory accesses are required on average and 18 cache memory accesses in the worst case. This shows that the proposed algorithm can efficiently cut the search range down with the index table at the cost of one or two cache memory accesses.

Table 3: Number of cache memory accesses to search small search groups

Size of small search group	Mae-East (%)	Mae-West (%)	AADS (%)	PAIX (%)
1	86.6	84.57	85.53	81.31
2~5	11.85	13.46	12.64	16.19
6~9	1.0	1.38	1.30	1.68
10~12	0.32	0.35	0.31	0.58
13~15	0.18	0.19	0.18	0.21
16	0.05	0.05	0.04	0.02
# of average accesses	0.43	0.50	0.47	0.62
# of worst case accesses	16	16	16	16

4.2 Memory usage

Since the index table has a fixed size of 2.25 Kbytes, independent of the size of the forwarding tables, the majority of the memory usage for this algorithm will be due to the hash table and the small search groups. The results of simulation shows that the memory usage increases gradually with the hash table size. With a fixed hash table size of 52000 elements, the memory usage for the four databases is between 700 Kbytes and 1M bytes. This implies that a larger hash table does not consume too much memory space. Therefore within the memory constraints, the hash table size should be as large as possible to improve the hash lookup performance.

4.3. Updates

In Section 3.4, we have shown that updating a prefix will involve 1 to 18 main memory accesses. However, if the length of this prefix is less than the length of the hash key, the prefix needs to be extended to $2^{(k-p)}$ prefixes before being updated, where k and p are the length of hash key and the prefix, respectively. Thus, we must consider the number of extended prefixes to evaluate the updates. Table 4 lists the results of main memory accesses and extended prefixes.

Table 4: Implementation results of updates

	Mae-East	Mae-West	AADS	PAIX
Average case	6.09	7.26	6.75	4.13
Worst case	356	346	354	324

Although updating a new prefix may create several extended prefixes on average and 128 prefixes in the worst case, the number of main memory accesses is still kept between four and eight. This shows that most of the extended prefixes belong to the small search groups; so, updating a prefix only requires less than eight main memory accesses on average case and around 350 main memory accesses in the worst case.

5. Conclusion

We have proposed a novel IP address lookup algorithm that combines a statistical analysis of the forwarding table to save memory to a fast hash-based lookup for speed. By employing three specifically designed data structures, index table, hash table and small search group, the search range can be quickly be narrowed down from the whole lookup table to a small search group, which significantly speeds up the lookup rates. Using only one hash table not only limits the memory requirements, but also reduces the complexity of the algorithm. The small search group limits the memory modifications in the prefix updates, which allows fast prefix updates.

The implementation of the algorithm on a general purpose processor shows that finding the longest matching prefix for a given IPv4 address requires one main memory access and two cache memory accesses on average. To establish the lookup tables in this proposed algorithm, the memory requirements are between 700 Kbytes and 1Mbytes for four databases Mae-East, Mae-West, AADS and PAIX; updating a prefix involves less than eight main memory accesses on average.

Reference

[1] Internet Engineering Steering Group, "Applicability Statement for the Implementation of Classless Inter-Domain Routing (CIDR)", RFC 1517, Available at http://www.itef.org/rfc/rfc1517.txt, 1993.

[2] D.E. Knuth, "Optimum Binary Search Trees", Acting Information, 1971, pp14-25.

[3] D.R. Morrison, "PATRICIA — practical algorithm to retrieve information coded in alphanumeric", Journal of the ACM, Oct.1968, pp. 514-534.

[4] K.Sklower, "A tree-based packet outing table for Berkeley Unix", In Proceedings of 1991 Winter USENIX Conference, Dallas,1991, pp. 93 –99.

[5] Stefan Nilsson and Gunnar Karlsson, "IP Address Lookup using LC-Tries", IEEE Journal on Selected Areas in Communications, June 1999, pp. 1083-1092

[6] V.Srinivasan and G.Va ghese, "Faster IP lookups using controlled prefix expansion", Proceedings of SIGMETRICS 98, Madison, 1998, pp. 1-10

[7] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable high speed IP routing lookups", in *Proc. ACM SIGCOMM'97*, Sept. 1999, pp. 25-35.

[8] Andreas Moestedt, Peter Sjodin: "IP Address Lookup in Hardware for High-Speed Routing", *Proc. IEEE Hot Interconnects 6 Symposium,* Stanford, California, USA, August 1998, pp. 31-39

[9] P. Gupta, S. Lin and N. McKeown, "Routing lookups in hardware at memory access speeds", in *Proc. IEEE INFOCOM*, vol. 3, Mar./Apr. 1998, pp. 1240-1247.

[10] Derek Pao, Cutson Liu, Angus Wu, Lawrence Yeung nd K. S. Chan, "Efficient hardware architecture for fast IP address lookup", IEEE INFOCOM 2002, June 2002, pp. 555-561.

[11] Kari Seppänen, "Novel IP address lookup algorithm for inexpensive hardware implementation", WSEAS Transactions on Communications, Vol.1, No.1, 2002, pp. 76-84.

[12] A. McAuley and P. *Francis,* "Fast routing table lookup Using CAMS", In Proceedings of INFOCOM, March-April 1993, pp 1382-1391.

[13] Tzi-cker Chiueh and P. Pradhan, "High-Performance IP Routing Table Lookup Using CPU Caching", IEEE INFOCOM, 1999, pp. 1421-1428.

[14] C.P.et al, "A 50-Gb/s IP router", IEEE/ACM Transactions on networking, 1998, pp.237-248

[15] David E. Taylor, John W. Lockwood, Todd Sproull, Jonathan S. Turner and David B. Parlour, "Scalable IP Lookup for Programmable Routers", In Proceedings of IEEE INFOCOM, 2002.

[16] Belenkiy Andrey, "Deterministic IP Table Lookup at Wire Speed", Available at http://www.isoc.org-/inet99/proceedings/4j/4j_2.htm. New Jersey Institute of Technology USA, July 2000.

[17] Marcel Waldvogel, "Fast Longest Prefix Matching: Algorithms, Analysis and Applications", PhD thesis, Swiss federal institute of technology, 2002.

[18] H.Y. Tzeng, "Longest prefix search using compressed trees", in Proc. IEEE Global Communication'98 Conf., Sydney, Australia.

[19] Henry H., Y. Tzeng and Tony Przygienda, "On Fast Address-Lookup Algorithms", IEEE Journal on Selected Areas in Communications, June 1999, pp. 1067-1082.

[20] Kari Seppänen, "Novel IP address lookup algorithm for inexpensive hardware implementation", WSEAS Transactions on Communications, 2002, pp. 76-84.

[21] M.A.Ruiz-Sanchez, E.W.Biersack, and W.Dabbous, "Survey and taxonomy of IP address lookup algorithms", IEEE Network Vol.15, March-April, 2001, pp. 8-23.

[22] B.Lampson, V.Srinivasan, and G.Varghese, "IP lookups using multi-way and multicolumn search", In Proceedings of IEEE Infocom'98, San Francisco, 1998, pp. 1248-56.

[23] M.Degemak, A.Bodnik, S.Calsson, and S.Pink, "Small forwarding tables for fast routing lookups". ACM Computer Communication,Oct.1997,pp.3-14

[24] "Internet Performance Measurement and Analysis Project", University of Michigan and Merit Network, Available at http://www.merit.edu/ipma, 2002.