A Fast IP Address Lookup Algorithm Based on Search Space Reduction

Hyuntae PARK†, Hyunjin KIM†, Hong-Sik KIM†, Nonmembers, and Sungho KANG(✉), Member

SUMMARY This letter proposes a fast IP address lookup algorithm based on search space reduction. Prefixes are classified into three types according to the nesting relationship and a large forwarding table is partitioned into multiple small trees. As a result, the search space is reduced. The results of analyses and experiments show that the proposed method offers higher lookup and updating speeds along with reduced memory requirements.

key words: IP address lookup, search space reduction, partitioning

1. Introduction

Due to a tremendous increase in internet traffic, routers should be able to forward massive incoming packets at several gigabits per second. IP address lookup determines the output port of incoming packets by looking up a destination IP address in the forwarding table. This is one of the most challenging tasks for routers. In order to avoid wasting address space with classful routing, classless inter-domain routing (CIDR) was deployed. CIDR allows for arbitrary aggregation of IP addresses so that the prefix has variable lengths from 8 to 32 bits. Therefore, IP address lookup becomes more complicated because it must find the longest matching prefix among multiple matching prefixes [1].

A natural way to find the longest matching prefix is using a radix tree (trie) [2]. The trie is a tree-based structure which executes a linear search on prefix length. Each prefix is associated with a node defined by the path from the root. However, in a trie structure, the memory space is wasted since many empty internal nodes are required. In order to perform a binary search in a tree which has no empty nodes, a comparison method for values with different lengths has been suggested in [3]. Using the comparison method, a binary search method for prefix values was developed in [4]. However, due to a nesting relationship dependent on a hierarchy of prefixes, these algorithms have a limitation that the ancestor (shorter) prefix should be searched earlier than the descendant (longer) prefix. Accordingly, even if a match occurs in the middle of a tree, a search should continue to a leaf for the best matching prefix.

In order to overcome this limitation, the longest prefix first search (LPFS) has emerged. In [5], empty internal nodes of a trie are replaced by the longest prefix among the descendent prefixes of the value associated with each empty node. In [6], the longer prefix is always located on the upper node in a trie, regardless of the nesting relationship. Among shorter prefixes, the prefix associated with the position in a trie overlaps with the previously allocated prefix. Both methods can search from the longer prefix in decreasing order of length. However, in order to allocate the longer prefix on the upper node in a trie at all times, recursive exchanges of nodes are required for the building and updating procedure.

On the other hand, in order to avoid the nesting relationship among prefixes, partitioning techniques have been suggested. In [7], all prefixes are partitioned into multiple tables based on their output port. However, since this scheme is processed in parallel, additional hardware is required. In [8], a forwarding table is divided into multiple trees which consist of a subset of ancestor prefixes. However, although the forwarding table is partitioned, the search space is not reduced.

In the proposed method, by considering the nesting relationship among prefixes, a large forwarding table can be partitioned into multiple small trees. Hence, the limitations of a binary search for IP address lookup are eliminated and the drawbacks of LPFS schemes are minimized. In addition, the proposed partitioning technique can reduce the search space so that a fast IP address lookup can be achieved.

2. Proposed Method

In this letter, in order to represent the nesting relationship of each prefix, the types of prefixes are defined as disjoint, parent, and child. A prefix is defined as disjoint if it has no other nesting or nested prefix. Otherwise, a prefix with a nesting relation is defined as a child. Among child prefixes, the shortest prefix is defined as a parent. Using this definition, Fig. 1 shows the relations among example prefixes.

As shown in Fig. 1, disjoint prefixes and parent prefixes

Fig. 1 The nesting relationship of example prefixes.
have no nesting relationship regardless of their length. Inspired by this fact, we carefully note that not all prefixes are necessary for longest prefix matching; only child prefixes are required. In the proposed method, disjoint prefixes and parent prefixes are separated from child prefixes. Then, disjoint and parent trees (DP-trees) are built from multiple binary search trees which are divided by the first 8 bit substrings of prefixes. Child trees (C-trees) consist of multiple LPFS based trees on each parent prefix. Based on multiple small trees, the search is restricted to within one DP-tree, or a part of one DP-tree and one C-tree. As a result, it is certain that the number of memory accesses in a lookup operation is less than \( \log_2 N \) with \( N \) entries.

2.1 Building

Prefixes are first classified based on their nesting relationship. Each prefix has its own type such as disjoint, parent and child. For the classified set of prefixes in Fig. 1, the proposed multiple trees are built as shown in Fig. 2. In this example, DP-trees are divided by the first 2 bits of prefixes instead of the first 8 bits. Disjoint prefixes and parent prefixes are entries of DP-trees. Since these prefixes are mutually exclusive, DP-trees can be built out of a balanced binary search tree with no consideration of the nesting relationship. Child prefixes are entries of C-trees. In order to overcome a limitation due to the nesting relationship among child prefixes, C-trees are built using the LPFS scheme of [6].

A search space reduction is achieved by three mechanisms as follows. First, DP-trees are divided into \( 2^8 \) multiple trees with the first 8 bit sub-strings of prefixes because there are no prefixes with lengths less than 8 bits. Therefore, the search space of DP-trees is chopped. Second, the parent prefixes are built earlier than the disjoint prefixes in DP-trees. These are the black nodes shown in Fig. 2. As a result, when packets are searched through DP-trees into C-trees, these are not compared with disjoint prefixes in DP-trees. Finally, in order to reduce the search space of C-trees, they are divided by each parent prefix. In Fig. 2, three C-trees are built about 1010, 100 and 111.

For the proposed multiple trees, the forwarding table is shown in Fig. 3. In the starting table, each entry has a pointer to the root of each DP-tree. The first 8 bits of prefixes are used as the index of the starting table. In this example, the first 2 bits of prefixes are used. These bits are not stored in the prefix field of DP-trees, and the remaining bits of the prefix are stored. Each entry of a DP-tree has a left pointer and a right pointer for a binary search. The entries of parent prefixes also have sub pointers and the number of entries which address their own C-trees. Otherwise, each entry in a C-tree has a superior output pointer instead of the fields of the sub pointer and the number of entries. In the LPFS scheme, the existing entry allocated to a node may contain the prefix associated with the position in a trie. The existing entry for a node is stored in the fields of C-trees. The output pointer of the contained prefix is stored in the field of the superior output pointer. This overlapped entry is shown as the white node in Fig. 2 and 100101’s entry containing 1001’s output pointer in row 23 in Fig. 3. The field for the superior output pointer of the prefixes is not shown in Fig. 3. The gray areas in Fig. 3 represent the free space for the prefix insertion. Because each DP-tree is indicated by the entry in the starting table, the free space for DP-trees is partitioned so that it is allocated at the end of each DP-tree. Otherwise, the free space for C-trees is the remaining space at the end of the forwarding table.

2.2 Searching

First, using the first 8 bits of the incoming address as the index of the starting table, the DP-tree is selected and a binary search is performed. If a matched disjoint prefix is found, the search is immediately finished and the output pointer of the matched prefix is returned.

On the other hand, if a matched parent prefix is found, the search is continued into the associated C-tree and the output pointer of a matched prefix is temporarily stored as the best match prefix (BMP). In the C-tree, the search procedure is the same as in a binary trie based LPFS scheme. The search proceeds to the left or right according to the sequential check of address bits. If entries with the superior output pointer are visited, BMP is changed along with the superior output pointer. If a matched prefix is found in the C-tree, the search is completed and the output pointer of the matched prefix is returned. Otherwise, BMP is finally re-
turned as the output pointer.

In an example in Fig. 3 for the incoming address of 100110, 10’s DP-tree is selected and the search space in DP-trees is from 10 to 12 in the forwarding table. In this space, the matched prefix is 100. Since it is a parent prefix, the 100’s output pointer is stored as BMP and the search is continued into 100’s C-tree. The incoming address, 100110 is not matched with 100100, which is the root of 100’s C-tree. Next, in 100110, the first bit excluded from 100 is 1, so the search is compared with 100101. It is not matched, but BMP is changed with the superior output pointer, which is 1001’s output pointer. Then, since the next bit is 1, the search is completed. Because there is no final matched entry, 1001’s output pointer as the final BMP is returned. Otherwise, in another example, if the incoming address is 10010, 10010’s output pointer is returned regardless of the BMP.

2.3 Updating

The proposed method supports incremental updating. When a new prefix is inserted, there are three cases in which the trees are modified. First, if the new prefix is not matched with any entry in the DP-trees, it becomes a new DP-tree entry as a disjoint prefix. The new prefix is simply inserted into the next empty node in the free space of the corresponding DP-tree. In the second case, if the new prefix is the substring of an existing DP-tree entry, it becomes the newest entry of the DP-trees as a parent prefix. At the same time, if the existing entry was a disjoint prefix, this is a new entry of the C-tree which is newly generated. At this moment, a memory copy occurs only once. On the other hand, if the existing entry was a parent prefix, this is included in corresponding C-trees. In the third case, when the new prefix is matched with the existing parent prefix in DP-trees, it is included with the corresponding C-tree as a new child prefix. The new entry is inserted into the free space for C-trees, or is contained within the existing prefix. The modification of C-trees is only performed by updating the left and the right pointer of nodes.

3. Performance Analysis

When \( N \) is the total number of original prefixes, \( D, P \), and \( C \) denote the number of disjoint, parent and child prefixes, respectively, so that \( N = D + P + C \). In addition, we set \( DP = D + P \), which is the total number of entries in DP-trees. \( C' \) is the number of child prefixes reduced by using the LPFS scheme, and it is the total number of entries in C-trees. Hence, we set \( N' = DP + C' \) to be the total number of entries in the proposed method. Besides, the distribution of prefixes over the first 8 bit sub-strings is assumed to be uniform. The distribution of child prefixes over each parent prefix is assumed to be uniform as well.

The most crucial metric for IP address lookup schemes is lookup speed. Because the cost of computation is dominated by memory accesses, lookup speed can be evaluated by the worst-case number of memory accesses. The worst-case number of memory accesses for a binary search-based IP lookup scheme depends on the worst-case tree depth.

The worst-case tree depth of the proposed method may exist in DP-trees or C-trees, according to the number of entries in DP-trees and C-trees. When \( DP < C' \), the worst depth lies in DP-trees. Since DP-trees are divided into \( 2^8 \) multiple trees, the depth is calculated as \( \log_2(DP/2^8) \). On the other hand, when \( DP > C' \), the worst depth lies in C-trees along with a part of DP-trees. Since parent prefixes are built prior to disjoint prefixes, the depth of a part of DP-trees is \( \log_2(P/2^8) \). C-trees are constructed based on a trie with no empty internal nodes on each parent prefix, so that the depth of C-trees is \( \log_2(C'/P) \). Therefore, the worst depth is calculated by the sum of \( \log_2(P/2^8) \) and \( \log_2(C'/P) \), that is \( \log_2(C'/2^8) \). In conclusion, the depth of trees in the proposed method is

\[
\log_2 \frac{DP}{2^8} \text{ or } \log_2 \frac{C'}{2^8},
\]

(1)

Table 1 compares the performance of the proposed method with that of other algorithms. Let \( N, L \) and \( L_{op} \) be the total number of prefixes, the depth of the prefix tree and the optimum depth, respectively. In previous binary tree based lookup methods, the depth is \( L \geq \log_2 N \). Since \( L_{op} \) should be an integer, \( L_{op} = \lceil \log_2 N \rceil \). For the IFPLUT scheme [7], \( F \) denotes the number of partitions for parallel processing. In LPFS [6], the entries are reduced from a whole prefix set. However, in the proposed method, the entries are only reduced from a child prefix set. Therefore, the number of entries in LPFS (\( N'' \)) is less than that in the proposed method (\( N' \)). In addition, in order to compare \( DP \) and \( C' \) with \( N'' \), we have examined various real routing data. As a result, it is absolutely true that

\[
DP & C' < N'' < N' < N.
\]

(2)

Consequently, the search space can be reduced from \( N \) to \( DP \) or \( C' \), so that the lookup speed of the proposed method can be significantly increased compared to previous methods.

Memory requirements are evaluated by multiplying the node size by the number of entries. Although the node size required by the proposed method is not the smallest, the reduction of entries conserves the memory resources so that the required memory size of the proposed method is reasonable.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Lookup speed ((L_{op}))</th>
<th>Memory usage</th>
<th>Node size (bytes)</th>
<th>entries #</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPT[3]</td>
<td>([\log_2 N])</td>
<td>N</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>WBPT[4]</td>
<td>([\log_2 N])</td>
<td>N</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>P-Trie[5]</td>
<td>([\log_2 N])</td>
<td>9</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>IFPLUT[7]</td>
<td>([\log_2 N/F])</td>
<td>11</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MBT[8]</td>
<td>([\log_2 N])</td>
<td>8</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Proposed method</td>
<td>([\log_2(DP/2^8)) )</td>
<td>or ([\log_2(C'/2^8)])</td>
<td>10</td>
<td>(N')</td>
</tr>
</tbody>
</table>
4. Experimental Results

The performance evaluations of the proposed algorithm are simulated using C language based on various routing data from real backbone routers. Table 2 shows the experimental comparisons with other algorithms for several metrics such as lookup speed, updating speed and memory requirement. Because the operation speed is dominated by memory accesses, the lookup speed is measured by the average and the maximum number of memory accesses. The updating speed is determined by the number of node changes, compared with previous LPFS based algorithms.

As shown in Table 2, the lookup speed of the proposed method is the fastest in the average and worst cases. The proposed method can rapidly complete the matching operations by dealing with the nesting relationship among prefixes, so that the average number of memory accesses can be reduced. In addition, because the binary trees, which are used in the previous schemes, are not perfectly balanced, the worst depth is greater than the optimum depth. Therefore, the worst-case lookup speeds of the previous methods depend on a balance degree of trees. Since the proposed method generates and applies more balanced trees than previous methods, the worst-case memory accesses of the proposed scheme can be the smallest. Although the experimental results depend on the routing data that is originally input, the updating speed is generally good. When the number of entries is small, as in Mae-West, the updating speed is remarkably improved. Because the node sizes of P-Trie and MBT are smaller than that of the proposed method, the required memory sizes of P-Trie and MBT are smaller in Mae-West. However, when the number of entries is large, as in Port80 and Telstra, the required memory size of the proposed method is relatively less.

5. Conclusion

In this letter, a new fast IP address lookup algorithm is proposed based on search space reduction. The proposed method splits the large forwarding table according to the types of prefixes so that the search space is efficiently reduced. The performance analysis and experimental results show that the proposed method yields excellent performance in terms of lookup speed, updating speed and memory requirements.

References