IP Address Lookup for Internet Routers Using Balanced Binary Search with Prefix Vector

Hyesook Lim, Member, IEEE, Hyeong-gee Kim, Student Member, IEEE, and Changhoon Yim, Member, IEEE

Abstract—We propose an efficient binary search algorithm for IP address lookup in the Internet routers. While most of the previous binary search algorithms do not provide a balanced search, the proposed algorithm provides a perfectly balanced search, and hence it provides excellent search performance and scalability toward large routing tables.

Index Terms—Internet, router, IP address lookup, binary trie, binary search tree.

I. INTRODUCTION

INTERNET routers need to perform packet forwarding in high speed, and IP address lookup is one of the most challenging tasks. The longest prefix matching operation for the IP address lookup is an important issue due to the rapid growth of the Internet traffic and the scalability toward IPv6. The current solution for IP address lookup using ternary content addressable memory (TCAM) is confronted by the scalability issue, and efficient algorithmic solutions are essentially required. As a low-cost solution, binary search algorithms such as a binary trie or a binary search tree have been popularly used. However, the binary trie involves many empty internal nodes, which cause the waste of memory and the deteriorated search performance. The binary search tree is a better approach in the sense that it does not involve empty internal nodes. However, the prefix nesting relationship, in which a prefix is a sub-string of another prefix, makes the tree unbalanced, and hence it does not provide high speed search.

In this letter, we propose a new balanced binary search algorithm based on the binary search tree. To construct a balanced binary search tree, we propose to remove the prefix nesting relationship of nodes in the binary search tree and make each node hold a prefix vector containing the nesting information.

II. RELATED WORKS

We first define the IP address lookup problem formally as follows. Let \( \mathcal{P} = \{P_1, P_2, \cdots, P_N\} \) be the set of routing prefixes where \( N \) is the number of prefixes. Let \( A \) be an incoming IP address and \( S(A, k) \) be a sub-string of most significant \( k \) bits of \( A \). Let \( n(P_i) \) be the length of a prefix \( P_i \) in bits. \( A \) is defined to match \( P_i \) if \( S(A, n(P_i)) = P_i \).

Let \( \mathcal{M}(A) \) be the set of prefixes that \( A \) matches, then \( \mathcal{M}(A) = \{ P_i \in \mathcal{P} : S(A, n(P_i)) = P_i \} \). The longest prefix matching problem is to find the prefix \( P_j \) in \( \mathcal{M}(A) \) such that \( n(P_j) > n(P_i) \) for all \( P_i \in \mathcal{M}(A), i \neq j \). Once the longest matching prefix, \( P_j \), is determined, the input packet is forwarded to an output port directed by the prefix \( P_j \).

An example binary trie for an arbitrary set of prefixes is shown in Fig. 1. By examining each bit of an incoming address starting from the most significant bits, search proceeds to the left child or the right child until there is no more child to proceed. At each node, if the input matches the stored prefix, it is remembered as the current best matching prefix (BMP). As shown, empty nodes (white nodes), which are not associated with any prefix, are included in the paths going to prefix nodes (dark nodes) [1]. In order to remove empty nodes, the priority trie (P-trie) [2] relocates the longest prefix of the sub-trie rooted by an empty node into the empty node. For example, in Fig. 1, the prefix \( P_3 \) can be relocated to the root node, the prefix \( P_4 \) can be relocated into the node of \( 11^* \), and so on.

With this simple change, the priority trie reduces the depth of the trie and provides faster search speed than the binary trie. The well-known level-compressed trie (LC-trie) [3] applies both the level compression and the path compression to the binary trie. The LC-trie significantly reduces the depth of the trie and it is effective in search performance, but it requires huge memory space because of empty nodes created newly in the process of the level compression.

To perform the binary search for prefixes with different lengths, prefixes should be primarily sorted in the order of magnitudes. In the binary search tree (BST) algorithm [4], a set of new definitions are provided to sort the prefixes with

Authorized licensed use limited to: National Cheng Kung University. Downloaded on April 19, 2009 at 09:19 from IEEE Xplore. Restrictions apply.
different lengths as follows. For two prefixes, $P_i$ and $P_j$ with $n(P_i) < n(P_j)$, if $P_i > S(P_j, n(P_i))$, $P_i$ is bigger than $P_j$, vice versa. If $P_i = S(P_j, n(P_i))$, then we call the prefix $P_i$ is nested in the prefix $P_j$, and the prefix $P_i$ is defined as an enclosure prefix and the prefix $P_j$ is defined as an enclosed prefix. In this case, the $(n(P_i)+1)$-th bit of $P_j$ is compared. If it is 0, $P_j$ is defined bigger than $P_i$. Otherwise, $P_j$ is defined bigger than $P_i$. For the example of two prefixes, $00^*$ and $010^*$, the prefix $010^*$ is bigger than $00^*$ by comparing the first two bits. For two prefixes, $1^*$ and $1101^*$, $1101^*$ is bigger than $1^*$ since the first bit is the same and the second bit of the longer prefix is 1. If we scan the binary trie in Fig. 1 from the left to the right, we have the same list as sorted using these definitions.

However, the binary search cannot be directly applied to the list of prefixes sorted by these definitions. For example, assume that the sorted list is $00^*$, $1^*$, $1101^*$, $1101^*$, and $111^*$, and the input address is $110100^*$. Since the prefix $110101^*$ is in the middle of the sorted list, the input is compared with it. The input is smaller than the prefix $110101^*$, and hence the search proceeds to the left of the prefix $110101^*$. Therefore, the best matching prefix of the input address, which is the prefix $1101^*$, cannot be found. This failure comes from the existence of prefix nesting relationship and the enclosed prefix being compared earlier than the enclosure prefix.

To perform the binary search properly, the BST algorithm strictly restricts that the enclosure prefix is stored in a higher level than its enclosed prefix so that the enclosure is compared earlier, and this restriction makes the binary search tree highly unbalanced in case that there are many levels of prefix nesting.

Weighted binary search tree (WBST) considers the number of enclosed prefixes in selecting a root in each level so that it constructs a better balanced binary search tree [5]. However, it still fails to provide a perfectly balanced binary search. As an attempt to perform the balanced binary search, binary search on range (BSR) algorithm [6] considers each prefix as a range, which has a fixed-length start address and a fixed-length end address in a line of $[0, 2^{32} - 1]$. Each entry of a routing table is either the start address or the end address of disjoint ranges, and the balanced binary search is performed for those entries. However, the BSR algorithm requires complicated pre-computation as well as the number of entries of the BSR can be twice the number of prefixes in the worst case. A primary motivation of this letter is to provide an algorithm performing a perfectly balanced binary search without holding the number of routing entries more than the number of prefixes.

### III. Proposed Algorithm

We first define a prefix vector which contains prefix nesting information as follows. In Fig. 1, let $P_i$ be a leaf prefix of the binary trie such as $P_0$, $P_1$, $P_3$, $P_4$, or $P_5$. A prefix vector is composed of $n(P_i)$ elements as $V = [v_1, v_2, \ldots, v_{n(P_i)}]$. If $S(P_i, k) = P_i$ for $k = 1, \ldots, n(P_i)$, $v_k = Y_i$ where $Y_i$ is the output port, to which the input is forwarded when the prefix $P_i$ is the best matching prefix. If there is no $P_i$ such that $S(P_i, k) = P_i$, $v_k$ is a null element and is represented as $\phi$. The prefix vector does not need to compute the complete prefix value for nested prefixes, but just its length. The nested prefix is obtained by considering the correct number of bits of a leaf prefix. For the example in Fig. 1, the prefix vectors of leaf prefixes $110100^*$ and $11111^*$ are $[Y_2, \phi, \phi, Y_5, \phi, Y_3]$ and $[Y_2, \phi, Y_6, \phi, Y_7]$, respectively.

Our proposed algorithm is to construct a binary search tree only with leaves of the binary trie and make each leaf hold the nested prefix information using a prefix vector. The proposed binary search tree is shown in Fig. 2 for the same set of prefixes. Note that the number of entries in the proposed algorithm is smaller than the actual number of routing prefixes. Since each leaf prefix is free from nesting, the restriction coming from the prefix nesting relationship in building the binary search tree is not applied. Hence, the constructed binary search tree is perfectly balanced. Building the routing table of the proposed algorithm has two steps. First, a prefix vector is constructed for each leaf prefix of the binary trie. Second, leaf prefixes with a prefix vector are sorted in the ascending order and stored into a memory array. We do not need to store pointers for child nodes since a child node is always the medium entry of either the smaller or the bigger half of the sorted array in a balanced tree.

Let $V(u)$ be the prefix vector of a node $u$ and $P(u)$ be the leaf prefix stored at the node $u$. Let $K^*$ represent the best matching prefix length (BMPL) and $O^*$ represent the corresponding output port of the BMP. The binary search is performed as follows. The node $u$ is initially set as the root node, $K^*$ is set to zero, and $O^*$ is set to the default output port, to which the input is forwarded when there is no matching prefix.

```plaintext
do
  if $(S(A, n(P(u))) = P(u)) \Rightarrow$ input matches the leaf prefix
    $O^* = v_{n(P(u))};$ break;
  else
    for $k ← (n(P(u)) - 1)$ downto $(K^* + 1)$
      if $(v_k ≠ \phi \text{ and } S(A, k) = S(P(u), k))$ //input matches $P(u)$ in first $k$ bits
        $K^* = k; O^* = v_k;$ break;
      if $(A < P(u))$
        $u_\ell$ is the left child node of $u$;
      else
        $u_r$ is the right child node of $u$;
        $u ← u_r$;
    while ($u$ is a valid node);
  return $O^*$;

If the input $A$ matches the prefix of a leaf node $u$, i.e., $S(A, n(P(u))) = P(u), P(u)$ is the BMP. The proof of this
```

![Fig. 2. Proposed binary search tree with prefix vector.](image-url)
For the example of an incoming address, 111110, the input is compared with the leaf prefix stored in the root node which is 1101010. The input does not match the prefix but matches the first two bits. The v₂ is a null element and the v₁ is the output port, Y₂, and hence the BMPL is set to 1 and the output port is set to Y₂. Since the input is bigger than the stored prefix, search proceeds to the right child of the root node. The input is compared with the stored leaf prefix which is 111110* and matches the prefix. Now the output port is replaced by v₅ which is Y₅. Since the input matches the leaf prefix, search is finished and returns the current output port. If an incoming address is 111010, in the first access, the BMPL and the output port are set to 1 and Y₂, respectively. In the second access, the BMPL and the output port are replaced by 3 and Y₆, respectively. It is not updated in the third access, and hence Y₆ is returned.

The incremental update of the proposed algorithm is not simple, but it is possible. As in BSR [6], a prefix covers a range, in which the start and the end of the range is the prefix padded to the maximum length by zeros and by ones, respectively. In inserting or deleting a prefix, by searching both the start point and the end point, we can narrow down the range affected by the change. If the start and the end are the same, by examining one entry up or down, we can find out the longest prefix of a new entry or an obsolete entry, and hence we can handle the required prefix vector.

The entry width of the proposed routing table for IPv4 is maximum 30 bytes; 4 bytes for a leaf prefix, 1 byte for the leaf prefix length, and 25 bytes for the prefix vector. The prefix vector has 1 byte each for the output port of a nested prefix from length 8 to 32, considering that the shortest prefix length is 8 bits. Since a single cache line in general-purpose computing devices is usually more than 30 bytes, a routing entry of the proposed algorithm fits into a cache line. In case of IPv6, we can use the concept of prefix grouping, which makes a single prefix length represent a group of several prefix lengths [7].

In summary, the search complexity of the proposed algorithm is obviously $O(\log N)$. The update complexity is $O(2 \log N + \alpha)$, where $\alpha$ is the number of linear searches for entries in the affected range. The maximum number of routing entries is $N$, and each routing entry has a prefix vector, of which the width is proportional to $W$. Hence, the space complexity is $O(WN)$. 

### IV. Performance Evaluation and Discussion

The performance of the proposed algorithm is evaluated using various sizes of real routing tables downloaded on May 2006 from [8]. Since memory access is the most time-consuming operation in search process, we evaluated the search performance in terms of the number of memory accesses. The performance evaluation results are shown in Table I in terms of the number of prefixes ($N_{p}$), the number of leaf prefixes ($N_{f}$), the leaf rate, the average number of memory accesses ($T_{av}$), the maximum (worst-case) number of memory accesses ($T_{max}$), and the required memory size in storing the routing table ($M_{p}$). We also show the average memory consumption per prefix ($M_{p}$). As shown in Table I, the average number of memory accesses is 13 to 17. The maximum number of memory accesses is also bounded by a small number. Especially, the ratio of the number of leaves to the number of prefixes is relatively small for large routing tables, and hence the average and the maximum number of memory accesses are moderately increased as the growth of the routing tables. The required memory size is 15 to 24 bytes per prefix. The simulation result in Table I shows that the proposed algorithm is effective for large routing tables and hence provides good scalability.

Table II shows the performance comparison with other binary search algorithms. In the average number of memory accesses, \textit{\textbf{TABLE I}}

<table>
<thead>
<tr>
<th>Routing table</th>
<th>MacWest</th>
<th>MacEast</th>
<th>Port80</th>
<th>GroupItl</th>
<th>Telstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>14565</td>
<td>39464</td>
<td>112310</td>
<td>170601</td>
<td>227223</td>
</tr>
<tr>
<td>Leaf rate (%)</td>
<td>97.8</td>
<td>97.1</td>
<td>62.7</td>
<td>62.7</td>
<td>71.3</td>
</tr>
<tr>
<td>$T_{av}$</td>
<td>12.9</td>
<td>14.3</td>
<td>15.3</td>
<td>16.0</td>
<td>16.5</td>
</tr>
<tr>
<td>$T_{max}$</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>$M$ (kbyte)</td>
<td>348.8</td>
<td>935.5</td>
<td>1719.2</td>
<td>2611.3</td>
<td>3957.6</td>
</tr>
<tr>
<td>$M_{p}$ (byte)</td>
<td>24</td>
<td>24</td>
<td>15</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

\textit{\textbf{TABLE II}}

<table>
<thead>
<tr>
<th>Routing table</th>
<th>MacWest</th>
<th>MacEast</th>
<th>Port80</th>
<th>GroupItl</th>
<th>Telstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>14565</td>
<td>39464</td>
<td>112310</td>
<td>170601</td>
<td>227223</td>
</tr>
<tr>
<td>Leaf rate (%)</td>
<td>97.8</td>
<td>97.1</td>
<td>62.7</td>
<td>62.7</td>
<td>71.3</td>
</tr>
<tr>
<td>$T_{av}$</td>
<td>12.9</td>
<td>14.3</td>
<td>15.3</td>
<td>16.0</td>
<td>16.5</td>
</tr>
<tr>
<td>$T_{max}$</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>$M$ (kbyte)</td>
<td>348.8</td>
<td>935.5</td>
<td>1719.2</td>
<td>2611.3</td>
<td>3957.6</td>
</tr>
<tr>
<td>$M_{p}$ (byte)</td>
<td>24</td>
<td>24</td>
<td>15</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

...
accesses, the LC-trie is the best and the proposed algorithm is the next. In the memory consumption per prefix, LC-trie shows the worst performance and requires 3 to 27 times more memory than the proposed algorithm. The number of routing entries for P-trie, BST, and WBST is equal to the number of prefixes, and the width of the routing table is only 10 bytes. However, their search performance, especially in the maximum number of memory accesses, quickly deteriorates as the growth of the number of prefixes as shown. Note that the memory consumption per prefix of the BSR algorithm is small since the width of the routing entry is only 6 bytes even though the number of routing entries ($N_r$) is a lot more than the number of prefixes as shown. The number of routing entries of the proposed algorithm is the smallest, but the proposed algorithm requires a wide memory because of the prefix vector. However, if we consider the cache line size in the software-based implementation or the routing table stored using on-chip SRAM in the hardware-based implementation, we can utilize wide-word parallelism. Hence, it is better to reduce the number of entries rather than the width of the memory.

With respect to table update, LC-trie and BSR require very complicated data structure to provide the incremental update because of the prefix duplication or the pre-computation of the best matching prefix [9]. The P-trie, BST, and WBST provide the incremental update, and the maximum number of entries affected by a single change is equal to the maximum number of prefix nesting levels. Our proposed algorithm also provides the incremental update, but the number of entries affected by a change can be excessive. We can conclude that the proposed algorithm provides excellent search performance while requiring a reasonable amount of memory.

**APPENDIX**

Proof) It can be proven by contradiction. Since $u$ is a leaf node, there is no prefix node that has $u$ as its sub-string. In other words, there is no prefix $P(w)$ with $w \neq u$ in $P$ such that $n(w) > n(u)$ and $S(P(w), n(P(u)) = P(u)$. Assume that $P(u)$ is not the BMP. Then there exists a prefix $P(w)$ with $w \neq u$ in $P$ such that $n(w) > n(u)$ and $S(A, n(P(w)) = P(w)$. Since $S(A, n(P(u)) = P(u)$, $S(P(w), n(P(u)) = P(u)$. Hence there exists a prefix $P(w)$ with $w \neq u$ in $P$ such that $n(w) > n(u)$ and $S(P(w), n(P(u)) = P(u)$, which is a contradiction. Thus the matched leaf prefix $P(u)$ is the BMP.

**REFERENCES**


