# Parallel Processing Priority Trie-based IP Lookup Approach

Hootan Zhian, Muhammad Bayat, Maryam Amiri, Masoud Sabaei
Amirkabir University of Technology
Tehran, Iran
{hootan.zhian, m.bayat110, amiri.maryam, sabaei}@aut.ac.ir

***Abstract*—with the growth of Internet traffic and using Gb/s or 10 Gb/s links in backbone, the speed of forwarding packets in intermediate devices is crucial. Routers must be able to forward millions of packets per second on each of their interfaces. Finding a method that can speed up the IP lookup is one of the challenges in network research. There are two approaches to implement IP lookup, hardware-based and software-based. For software-based approach many algorithms have been proposed based on the tries concept. Some parallel algorithms have been suggested to exploit routers equipped with multi-core CPUs. In this paper, we propose a new approach that was constructed by combining the useful characteristics of the Priority trie and parallel processing. Our method consists of a trie and several subtries based on prefix length. These subtries processed by a specific thread to speed up IP lookup functionality. This proposed data structure inherits simplicity of Priority tries. Simulation results illustrate that our method speeds up IP lookup processing although memory usage has not been increased.**

***Index Terms— IP lookup, parallel processing, Priority trie***

## I. INTRODUCTION

Due to a tremendous increase in internet traffic today, backbone routers must have the capability to forward massive incoming packets at terabits per second and the speed of forwarding packets in routers is crucial. They must be able to forward millions of packets per second on each of their interfaces. IP lookup is the most time-consuming task in forwarding process. When a packet arrives in a router interface, it should decide to which interface the packet must be sent. Based on the packet destination IP address, routers find the appropriate next hop and the egress interface. Each router completes its routing/forwarding table based on the information it receives by routing protocol messages. Each router with the help of dynamic routing protocols propagates its knowledge of the network or topology modification through broadcast or multicast messages and could update its forwarding table based on this information. Based on the packet destination IP address, the router looks up the forwarding table to find the egress port. There are two types of forwarding table lookup, static and dynamic. In static forwarding table lookup, IP table is not updated regularly. The most significant disadvantage of static scheme is that a single addition or deletion of an entry may result in rebuilt of the entire table.

Based on this issue, dynamic strategy is mostly used today. Another issue that should be considered in routing is Classless Inter Domain Routing (CIDR). In regards of classful IPv4 address shortage, three solutions have been suggested to overcome this problem: NAT, IPv6 and CIDR. CIDR refers to arbitrary length of prefixes and address summarization at arbitrary levels. Routing table entries consist of pairs of (*p/l, o*), where p is the prefix, *l* is the length of the prefix which is at most 32 bit for IPv4 addresses and 128 bit for IPv6 addresses, and *o* is an output port. By using CIDR, the size of router tables will be decreased, but the IP table lookup operation becomes more complicated. The purpose of IP lookup and packet forwarding is finding longest matching prefix (LMP). Routers compare the destination IP address of an incoming packet with all the prefixes in routing table to find the LMP. For finding LMP, the router compares the destination IP address to all the prefixes in routing table, starting from most significant bit.

Many trie based algorithms have also been proposed [1-4]. These algorithms are wildly used software approaches. Routers, based on the trie and its data structure, decide how to find the LMP. In all trie algorithms, four main issues should be considered:

1. Memory space requirement: the data structure of the router table must be stored in memory so it must use less space to reduce the total size of the memory.
2. Lookup speed: the most significant issue in internet routers when forwarding a packet is the speed of finding an egress port.
3. Scalability: IPv6 will be the next wildly used layer 3 identifier, so algorithms should be able to apply to both IPv4 and IPv6.
4. Update Speed: route information changes frequently in Internet, and the speed of updating routing table by insertion and deletion of an entry is crucial.

In other approaches, scientists try to speedup IP lookup, based on hardware mechanisms. Many hardware-based algorithms have been suggested [5-8].



978-1-4799-5359-2/14/$31.00 ©2014 IEEE

Some approaches try to exploit both software and hardware-based algorithms. Speedup IP lookup, based on parallel processing, has also been suggested. Parallel Multiple Hashing Architecture [9] and DXR [10] are some examples. Parallel Multiple Hashing uses multiple hash tables and parallel processing, which DXR splits the large routing tables into the cache hierarchies of modern CPUs.

The rest of this paper is organized as follows. Section II reviews related works that are evaluated in the experiment. Section III introduces our lookup scheme, motivates of our design and its tradeoffs. In section IV we evaluate its performance and compare it to other algorithms. Section V is the conclusion and talks about the future studies.

## II. Related Works

Trie-based algorithms have been proposed to speedup IP lookup in routers. Binary trie [1] is a tree-based data structure. It uses linear search on length. In this algorithm prefixes will be placed in nodes and node level determines the prefix length. The search process starts from the root by inspecting from the most significant bit and travels to the left or right accordingly. Fig. 1, shows that Binary trie is inefficient in memory usage and search speed because there are many empty nodes in it. Also shorter prefixes are compared earlier than longer prefixes because they are placed at higher levels. In this algorithm when a match to an input address is found, the search process must continue to the leaves, maybe finding a longer prefix.

Srinivasan et al. [2] suggested a method to push prefixes to the leaves. For solving the problem of the longest prefix match in Binary trie, it should be transformed to Disjoint-prefix Binary trie. To obtain this trie, we simply push prefixes at intermediate nodes to leaves, by adding leaves to nodes that have only one or no child. These new leaves include new prefixes that inherit the bits of the closest granddad. Another benefit of this method is allocating separate memory size to each intermediate node and leaf. But this method is too costly for deleting or updating prefixes.

Hyesook Lim et al. [3] proposed Priority trie to replace empty nodes by the Priority prefixes which are longest among those in the subtrie rooted by the empty nodes. In this algorithm prefixes are sorted in the decreasing order of their length. From the beginning, the longest prefix is placed into the root node and the node is marked as Priority. Starting from the most significant bit of the next prefix, it will be placed in the left or right child of the root node and marked as a Priority. This process should be repeated for available prefixes. However, a prefix with length $|P|$ must be placed in a node in level $L$, which is less than or equal to $|P|$. If $L = |P|$, the node is marked as an ordinary node. The longest prefix match is selected by Priority encoder with hardware approach. The complexity in building the Priority trie is about $O(ND_p)$, where $D_p$ is the depth of the trie with N prefixes.

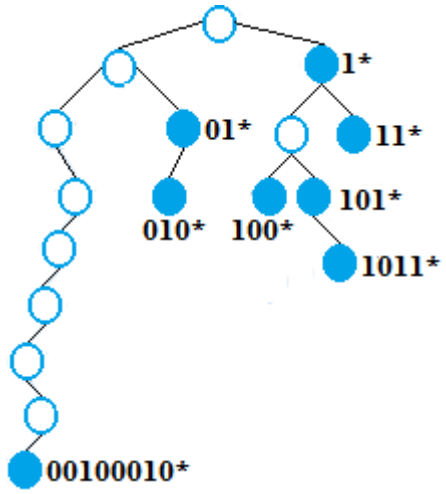



Fig.1. Binary trie.

The search process ends when an input matches the Priority prefix in any level without searching lower levels. When an input matches the prefix at an ordinary node, this prefix is stored as the current best match and the search continues. The search complexity is $O(D_p)$. This algorithm works well if prefixes are sparse. With the dense prefixes the trie becomes Binary trie. In the time of update, replacement will occur frequently. Another disadvantage is that, this algorithm does not consider prefix distribution. Address prefixes can be used to illustrate groups of contiguous addresses. Also algorithms must consider the number of prefixes and its distribution. Fig. 2, shows that 24-bit prefix length dominates the number of entries in the forwarding table. Furthermore, many hardware-based algorithms have been proposed to speed up IP lookup. DXR [10] is a hardware approach that splits the large routing table into the cache hierarchies of modern CPUs. Parallel Multiple Hashing [8] is another example which has been proposed by both software and hardware approaches. This method transforms IP Lookup table to some tables that are separated by the prefix length. Parallel IP lookup in each table is performed with the use of hashing. Hashing function is implemented with CRC. Hashing index is a pointer for each table. This algorithm decreases memory usage and the number of memory access and makes updating tables easy. But this is complicated because it combines software and hardware-based algorithms for doing parallel processing. We propose a new way to implement parallel processing in software-based approach with the help of multi-thread multi-core CPUs**.**

## III. The Suggested Algorithm

In this section we describe our proposed algorithm in detail. Before reviewing the detail of our data structure, we first introduce a trie structure named *Orderly* trie. Also modified bucket sort will be introduced for sorting prefixes. Then creation of the data structure will be discussed. Next, IP lookup and updating will be considered.

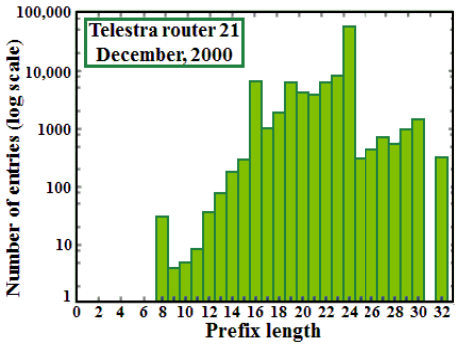



Fig. 2. Prefix length distribution of a typical backbone router

### A. Orderly trie

Let $P_i$ represents the prefixes, and $P = \{p_i | 1 < i < n\}$ is the set of prefixes in a router. $|P_i|$ represents the length of prefix $P_i$. *Priority* ($P_i$) shows the Priority of $P_i$. A Priority will be assigned to each entity in the *Orderly trie* and this Priority is equivalent to the Priority length, *Priority* $(P_i) = |P_i|$. Prefixes with shortest length have higher Priority. For example for prefixes $P_1$ and $P_2$ , $P_1$ has higher Priority if $|P_1| < |P_2|$. Like Priority trie, there are two types of nodes: Priority and ordinary.

### B. Constructing the trie

- At the beginning, prefixes must be sorted in the decreasing order of their lengths.
- Starting from a prefix with the highest Priority, the prefix is stored into the root node.
- Prefixes with the Priority less than the first one are stored in the left or right child of the root node based on the most significant bit.
- Other prefixes are added to the trie based on their Priority. Each prefix according to its bit-pattern (from most to the least significant bit) goes down until it reaches a leaf. Then according to its next bit (1 or 0) is stored in the left or right child of the previous node.
- A prefix with length $|P_i|$, must be placed in a node in level L, which is less than or equal to $|P_i|$. If L = $|P_i|$, the node is marked as an ordinary node.

The cost of constructing *Orderly* trie is similar to Binary trie. As prefixes are sorted according to their length, unlike Priority trie, there is no need to change the trie or reinsert a prefix in it.

Fig. 3 shows an *Orderly* trie of the Binary trie that shown in Fig. 1. Like Priority trie, *Orderly* trie eliminates empty nodes. The main differences between these two tries are the prefixes prioritization and the order they stored in the trie.

### C. Lookup in Orderly trie

- Based on the IP address bit pattern, we move down on the trie until we reach an ordinary node. Since the prefixes are stored based on their length, all prefixes before this ordinary one have shorter length.

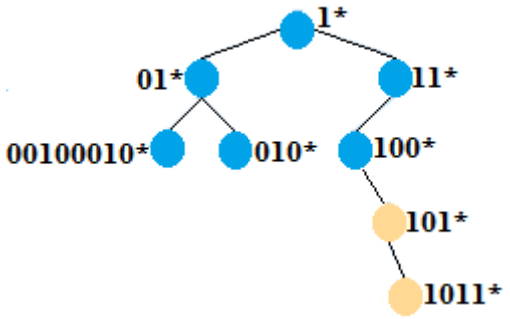



Fig. 3. Orderly trie

- We continue moving down from this ordinary node and compare the address with the prefixes in each node. If the address matches with a prefix, equivalent port is selected as an output port. This process continues until it reaches a leaf.

### D. Deletion

The process of deletion is easy as follows:

- First the node is deleted. If it had two children, the one with shorter length is replaces the deleted node. If the replaced node was an ordinary, it must be changed to Priority.
- If the deleted node had only one child, its child should replace it.

### E. Sorting prefixes

As mentioned before, both in Priority and Orderly trie, prefixes are prioritized and inserted into the trie based on their Priority. Sorting prefixes is an important issue here and the search complexity is $O(n \log n)$. If two prefixes have the same length, their Priority is the same and the one that first comes in address database is placed in the trie earlier. This is an important issue in suggested sorting method. Here we use *bucket sort* and with some changes, *Modified Bucket Sort* will be introduced.

### F. Bucket Sort

In this algorithm, elements that need to be sorted must be divided into multiple buckets and sorting (based on any algorithm) elements must be done in each bucket independently. The way that elements are placed in each bucket eliminates the need to sort elements between buckets so buckets are sorted.

### G. Modified Bucket Sort

In this algorithm, 32 buckets are defined based on the distribution of prefixes length. Prefix addresses are placed in an appropriate bucket based on their length. Unlike bucket sort, sorting elements in a bucket is not required, because elements in each bucket have the same length with the same Priority. Therefore, elements are stored in a bucket based on their place in the address database. The sorting complexity for each prefix is $O(1)$ and it is for determining a bucket for each

address. Therefore, the sorting complexity for n elements is $O(n)$.

*H. Suggested data structure*

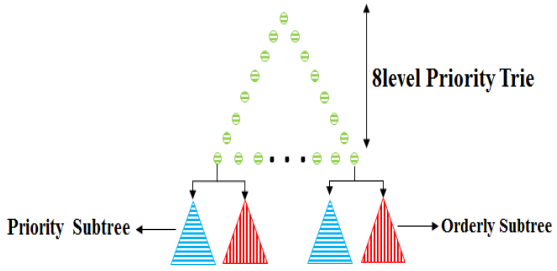
As mentioned before, our data structure is based on the distribution of the prefix length. Fig. 2 shows that how these prefixes are distributed. Our data structure is based on the following sections:

- 8 level Priority trie: a Priority trie with the maximum of eight levels to store 24 to 32-bit prefixes.
- Priority subtries: a set of Priority subtries that are defined from level 8. The remaining 24 to 32-bit prefixes are stored in these subtries. The subtrie ID is based on the first 8 bits of the prefixes.
- *Orderly subtries*: a set of *Orderly subtries* that are defined from level 8. Prefixes with 8 to 24 bits are stored in these subtries. The subtrie ID is based on the first 8 bits of the prefixes. Most of the prefixes are more than 8 bits. If prefixes with the length of 8 to 24 bits are stored in these subtries like Priority trie, most of the nodes may become ordinary and the Priority trie changes to Binary trie. Therefore, these subtries are *Orderly trie*. Fig. 4, shows our suggested data structure.

*I. Data structure creation*

For using parallel processing in constructing data structure, three threads must be used. Data structure construction steps are as follows:

- Threads start to read prefix addresses from address database with parallel processing. Each thread reads only 3K+ID lines based on their ID. (ID = {0,1,2} and k= 0,1,2,3,…)
- Based on the Modified Bucket Sort, each prefix that is read by a thread is stored in a linked list according to its length.
- At the end of reading prefixes, all linked lists that are created by these three threads are linked together.
- Addresses with length less than 24 bits are given to the first thread. This thread is responsible for inserting prefixes in *Orderly subtries*.
- Addresses with more than 24-bit length, are given to the second thread. If these prefixes cannot be inserted in Priority trie, they are given to the third thread and prefixes are inserted in Priority subtries. Fig. 5 shows the creation of the tree based on three threads.




- Fig. 4. Data structure of our algorithm

*J. Data structure creation pseudo code*

Fig. 5 shows data structure creation algorithm. The following pseudo code describe it:

1-1. Read $(3K)_{th}$ prefix lines.

1-2. Read $(3K+1)_{th}$ prefix lines.

1-3. Read $(3K+2)_{th}$ prefix lines.

2. Add prefix in $i_{th}$ linked list based on prefix length.

3. Merge three lists in one list.

4-1. Take prefixes with length of less than 24 bits, from smaller length to bigger one.

4-2. Take prefixes with length of more than 24 bits, from bigger length to smaller one.

4-3. If the Priority tree is completely full, the prefixes passed.

5-1. Insert prefix in one of subtries based on prefix 8 MSBs[1].

5-2. Insert prefix.

5-3. Insert prefix in Priority subtries based on prefix length between 8 to 24.

*K. IP address Lookup*

The process of IP lookup is as follows:

- The destination IP address is given to the mentioned three threads.

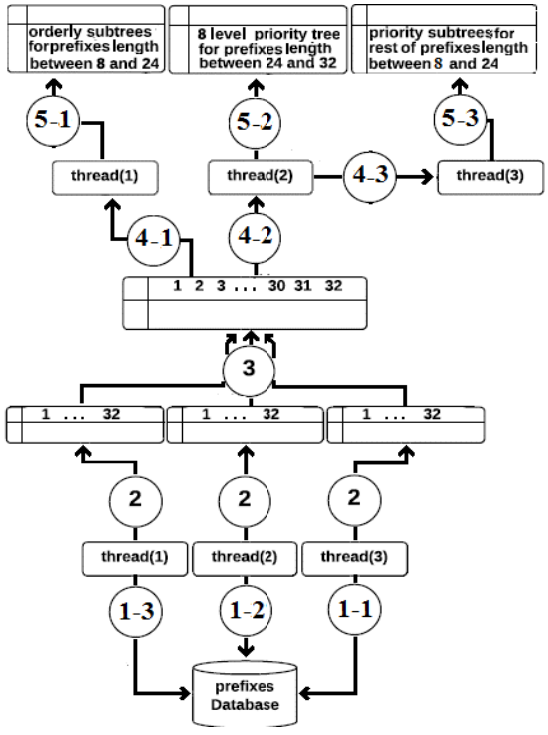



Fig. 5. Data structure creation algorithm

[1] Most Significant Bit

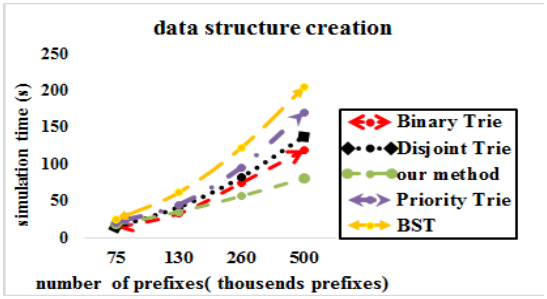
- Thread (1) in *Orderly* subtries, thread (2) in *Priority* trie and thread (3) in *Priority* subtries search for matching prefix in parallel.
- Thread (1) and thread (3) select the appropriate subtries based on the most significant 8 bits. They lookup these subtries by algorithms used in Priority and Orderly tries.
- Output port is selected based on the result that is given by a thread and longest prefix match.

## IV. Evaluation

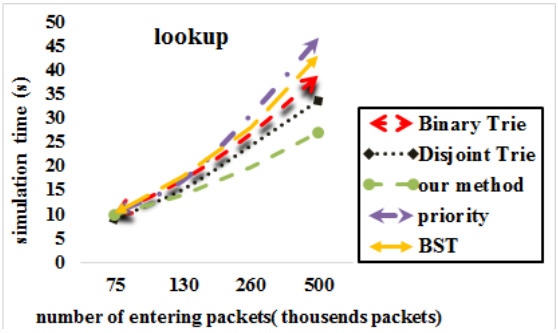
We implemented our method using C++ language in Linux and utilized Posix library which can be controlled and implemented in OS to implement threads. We evaluated the performance of our algorithm and compared it with Binary trie, disjoint trie, Priority trie and BST [11]. In the implementation, we evaluated the performance of algorithms in terms of data structure creation time, lookup time and memory consumption. We plotted the data structure creation time, lookup time and memory consumption respectively in Fig. 6(a), 6(b) and 6(c). As you can see in figure 1 Binary tree search method has better performance in smaller packet numbers. As you can see in Fig. 6(a), Binary trie has better performance in smaller prefix numbers. As number of prefixes increase, our method performance predominates the Binary trie. The reason is multithread characteristic of our method shows its beneficial role. Priority tire and BST have worst cost, because of data structure creation complexity. Fig. 6(b), shows our method has the lowest lookup time due to parallel processing and simple IP lookup approaches. Fig. 6(c), shows that our method, Priority trie and BST have the lowest memory consumption. As we showed, in our method the memory consumption order is O(1), so that each prefix only uses one node in our method data structure.

## V. Conclusion

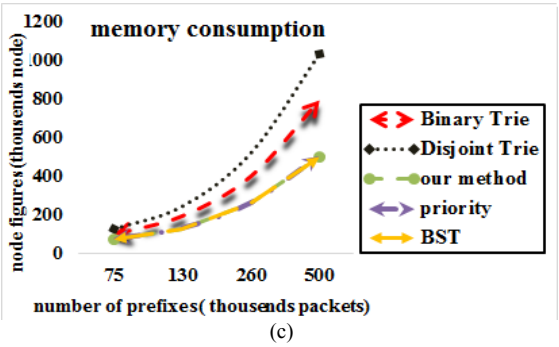
Lookup module is one of the most important functions that plays an important role in routers. Many software and hardware-based approaches have been proposed up to now. In this paper we utilized CPU's multicore feature and trie-based algorithms concurrently. We tried to optimize the Priority trie and suggested *Orderly trie*. In our suggested algorithm 24 to 32-bit prefixes are stored in the first 8 levels because these are the most used prefixes in the Internet. Priority and *Orderly subtries* were also used to store the remaining prefixes. Three threads and parallel processing were utilized to implement this algorithm. We compared our algorithm with Binary, Disjoint, Priority and BST. The simulation results showed that the proposed algorithm is superior in the time of constructing data structure, IP lookup time, update time and memory consumption.




(a)




(b)




(c)

Fig. 6. Simulation results, (a) Data structure creation time, (b) Lookup time, (c) Memory consumption


[2] V. Srinivasan, G. Varghese, "Faster IP lookups using controlled prefix expansion," *ACM SIGMATICS*, pp. 1–10, June 1998.

[3] H. Lim, C. Yim, E. E. Swart, "Priority tries for IP address lookup," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 784–794, Jun 2010.

[4] S. Hsieh, Y. Huang, Y. Yang, "Multiprefix Trie: A New Data Structure for Designing Dynamic Router-Tables," *IEEE Transactions on Parallel and Distributed Systems*, vol. 60, no. 5, pp. 693 – 706, May 2011.

[5] P. Gupta, S. Lin, N. McKeown, "Routing lookups in hardware at memory access speeds," *IEEE INFOCOM'98*, vol. 3, pp. 1240–1247, Apr 1998.

[6] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, Jan 2002.

[7] O. Erdem, A. Carus, H. Le, "Large-scale SRAM-based IP lookup architectures using compact trie search structures," *Computers and Electrical Engineering*, vol. 40, no. 4, pp. 1186–1198, May 2014.

[8] W. Jiang, V. K. Prasanna, "Sequence-preserving parallel IP lookup using multiple SRAM-based

pipelines," *Journal of Parallel and Distributed Computing*, vol. 69, no. 9, pp. 778-789, Sep 2009.
[9] H. Lim, Y. Jung, "A Parallel Multiple Hashing Architecture for IP Address Lookup," *IEEE,* pp. 91 – 95, 2004.
[10] M. Zec, L. Rizzo, M. Mikuc, "DXR: Towards a Billion Routing Lookups per Second in Software," *ACM SIGCOMM*, vol. 42, no. 5, pp. 30-36, Oct 2012.
[11] H. Park, H. Hong, S. Kang, "An efficient IP address lookup algorithm based on a small balanced tree using entry reduction," *Computer Networks*, vol. 56, no. 1, pp. 231-243, Jan. 2012.