Tree-Combined Trie: A Compressed Data Structure for Fast IP Address Lookup

Muhammad Tahir
Department of Computer Engineering,
Sir Syed University of Engineering and Technology,
Karachi

Shakil Ahmed
Department of Computer Engineering,
Sir Syed University of Engineering and Technology,
Karachi

***Abstract*—For meeting the requirements of the high-speed Internet and satisfying the Internet users, building fast routers with high-speed IP address lookup engine is inevitable. Regarding the unpredictable variations occurred in the forwarding information during the time and space, the IP lookup algorithm should be able to customize itself with temporal and spatial conditions. This paper proposes a new dynamic data structure for fast IP address lookup. This novel data structure is a dynamic mixture of trees and tries which is called Tree-Combined Trie or simply TC-Trie. Binary sorted trees are more advantageous than tries for representing a sparse population while multibit tries have better performance than trees when a population is dense. TC-trie combines advantages of binary sorted trees and multibit tries to achieve maximum compression of the forwarding information. Dynamic reconfiguration of TC-trie, made it capable of customizing itself along the time and scaling to support more prefixes or longer IPv6 prefixes. TC-trie provides a smooth transition from current large IPv4 databases to the large IPv6 databases of the future Internet.**

Keywords—IP address lookup; compression; dynamic data structure; IPv6

I. INTRODUCTION

Improvement of Internet-base multimedia applications in recent years drives new demands for high-speed Internet. It seems that the demand for achieving higher bit-rates never saturates. Having the fast optical fiber technology for data transmission, data processing elements, i.e. routers, became main bottleneck of the current Internet speed. Inside a router, components that limit its speed are IP address lookup and classification engines. The main role of router is to forward millions of packets per second on each of its destination by finding address of next-hop router or the egress port through which packet should be forwarded. This forwarding decision is limiting the speed as there are millions of addresses and finding destination IP from millions of IPs is not an easy task. There is a need to have an algorithm for efficient IP lookup. Before we go to details, let's see how IP addressing architecture works and evolving. Reviewing it will help us to understand the address lookup problem. IP addressing architecture can be divided into two schemes; classful IP addressing scheme and classless IP addressing scheme Classful IP scheme has two main issues; first, large number of IP addresses is wasted because of using IP address classes, second, the routing tables become very large. The growth of the forwarding tables resulted in higher lookup times and higher memory requirements in the routers and threatened to impact their forwarding capacity. In order to resolve two main issues there are two possible solutions one is IPv6 IP addressing scheme and second is Classless Inter-domain Routing or CIDR.

Finding a high-speed, memory-efficient and scalable IP address lookup method has been a great challenge especially in the last decade (i.e. after introducing Classless Inter-Domain Routing, CIDR, in 1994). In this paper, we will discuss only CIDR. In addition to these desirable features, reconfigurability is also of great importance; true because different points of this huge heterogeneous structure of Internet have different traffic shapes and network topology changes along the time at each point as well.

This paper proposes a new cost-efficient data structure for fast IP address lookup that dynamically reconfigures itself. This novel data structure combines binary sorted trees with variable-stride multibit tries and put advantages of them all together in itself.

A. Paper Organization

Section 1 explains importance and related work. Section 2 explores the idea of TC-trie by examples and explains how to build TC-trie from a binary trie. Section 3 shows the experimental results of the TC-trie implementation and finally Section 4 concludes the paper and reveals our future works.

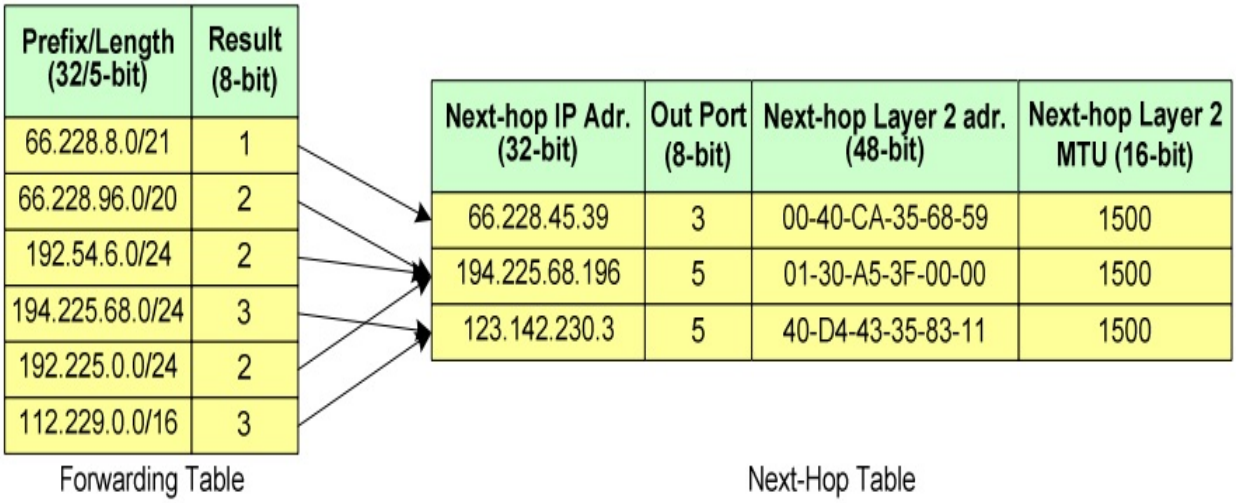
B. IP Address Lookup & Forwarding Tables

IP address lookup is a special search problem in a database of hundreds thousands of network addresses. Routers keep network addresses in their forwarding tables. For each incoming packet, the router finds a network address inside its table that matches with the destination IP address of that packet. Joint with each network address, there is a result field. The result could be simply an egress port of the router that packet should be exited from router via it to reach its destination network. Actually more information than an out port is required for forwarding a packet properly. This information includes next-hop layer-2 address, next hop layer-2 MTU (Maximum Transfer Unit), out port and so on. This information is kept in another table. This table can be called NHT (Next-Hop Table). Having the NHT, the result of the lookup could be a short length pointer to an entry of NHT. Fig. 1 shows an example of a forwarding table and an NHT. As this figure depicts, the forwarding table holds an 8-bit pointer corresponding to each network address.

A network address is a prefix of the 32-bit IP address. After introduction of CIDR (Classless Inter-Domain Routing) in 1994, network prefixes can be of any arbitrary length. In the classless routing, more than one network prefix may match with the destination IP address; in this case, the router must choose the longest prefix that matches; so the IP lookup problem is known as a Longest Prefix Match (LPM) problem.

C. Trees & Tries

During the last ten years, many solutions have been proposed that issue the LPM problem. Simply, the IP lookup solutions can be categorized as trie-based and tree-based methods. Fig. 2 shows trie and tree representations of a forwarding table in a supposedly 4-bit address space. In trees, information are hold explicitly in the nodes; so number of nodes is equal to the number of the network prefixes. If the binary tree to be balanced, its depth is ceiling [$\log_2$N] while N is the number of prefixes. In the opposite case, in tries, information are distributed on the edges. In a binary trie, each edge implicitly holds one bit of information. Left edges mean a zero bit and right edges mean a one bit. A path from the root to each node is a bit-string that corresponds to a network prefix. If this prefix exists in the table, the node should be labeled with corresponding result. Depth of a trie is equal to the length of the longest prefix that exists in the table.



Prefix/Length (32/5-bit)	Result (8-bit)
66.228.8.0/21	1
66.228.96.0/20	2
192.54.6.0/24	2
194.225.68.0/24	3
192.225.0.0/24	2
112.229.0.0/16	3

Forwarding Table

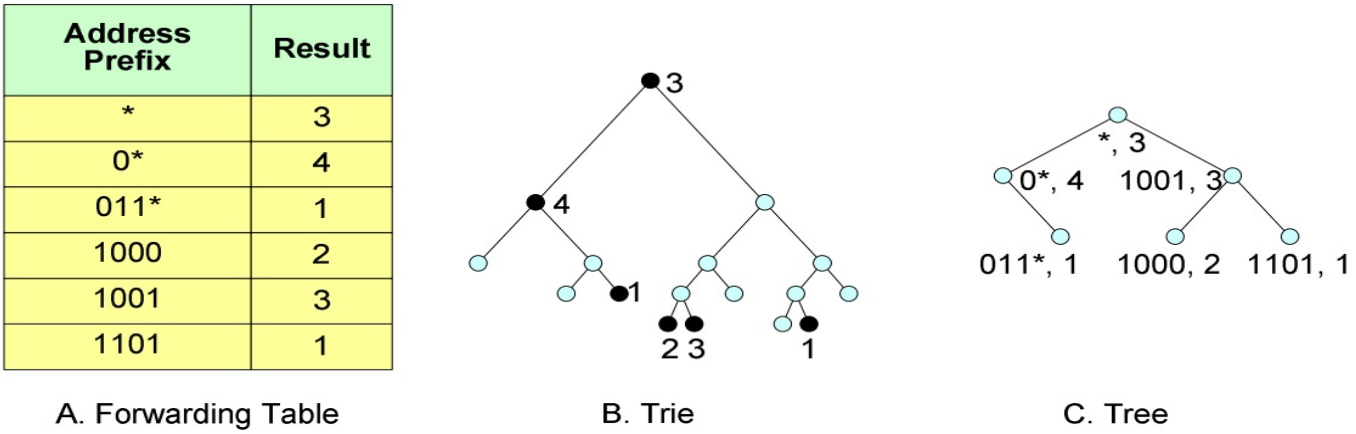
Next-hop IP Adr. (32-bit)	Out Port (8-bit)	Next-hop Layer 2 adr. (48-bit)	Next-hop Layer 2 MTU (16-bit)
66.228.45.39	3	00-40-CA-35-68-59	1500
194.225.68.196	5	01-30-A5-3F-00-00	1500
123.142.230.3	5	40-D4-43-35-83-11	1500

Next-Hop Table

Fig. 1. simple forwarding table pointing to a Next-Hop Table (NHT)

Latency of a lookup algorithm typically measured in the number of memory accesses that is equal to the depth of the data structure. Since depth of binary trees is less than depth of binary tries, they are faster than binary tries but they suffer from rebalancing overhead.

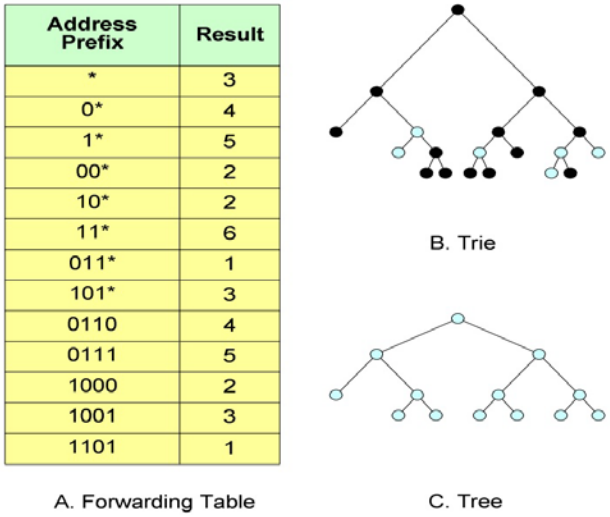
Memory consumption of trees and tries can be calculated by considering number of nodes in the data structure and size of each node. Number of nodes in tries is more than it in trees, because some nodes of the tries do not correspond to any valid prefix while in trees each node exactly keeps one prefix. The situation is different when considering the size of nodes. Since in the trees, prefixes are explicitly kept in the nodes,tree nodes are bigger than trie ones. Suppose that trie nodes are 32-bit while tree nodes are 64-bit. In the example of Fig. 2, memory consumption of the trie is 68 bytes while the memory consumption of the tree is 48 bytes; this means that in this example the tree is not only faster but also more compact than the trie. Fig. 3 shows another example for comparing trees with tries. In this example, the memory consumption of the trie is less than the one of the tree.



Address Prefix	Result
*	3
0*	4
011*	1
1000	2
1001	3
1101	1

A. Forwarding Table B. Trie C. Tree

Fig. 2. Trie and tree representations of a forwarding table. In this example, memory consumption of tree is less than trie



Address Prefix	Result
*	3
0*	4
1*	5
00*	2
10*	2
11*	6
011*	1
101*	3
0110	4
0111	5
1000	2
1001	3
1101	1

Fig. 3. Trie and tree representations of a forwarding table. In this example, memory consumption of tree is more than trie

When number of nodes in a trie is less than twice of number of tree nodes, trie consumes less memory than tree; in the other case, tree is more memory-efficient. As a rule of thumb, it can be concluded that trees are more compact than tries for sparse population of prefixes while when the prefixes are dense, tries are more compact.

D. Binary tries & Multibit Tries

Since a binary trie needs 32 memory accesses at the worst case for each address lookup, many solutions use multibit trie concept to accelerate the lookup search. In multibit tries, degree of nodes may be more than two. It means that an edge can hold more than one bit of information implicitly. Obviously, depths of multibit tries are less than binary tries and hence they are faster than binary tries. In fact, in multibit tries, the IP address segments into some strides. For example, the 32 bit IPv4 address can be segmented as 16-4-4-8. This segmentation corresponds to a four strides (or four levels) multibit trie with strides of length 16, 4, 4 and 8. The depth of a multibit trie is equal to the number of its strides. The memory consumption of a multibit trie depends to the number of strides and the length of each stride. Fig. 4 shows tree examples for comparing a tree, a binary trie and a one level multibit trie in a 4-bit address space. In the first example, the tree, in the second example, the trie, and in the third example, the multibit trie is the most memory efficient data structure. These examples demonstrate that the proper data structure should be chosen regarding the sparseness and the distribution shape of the prefixes in the address space. In general, it could not be said that trees are always more compact than tries or multibit tries consumes more memory than binary tries; the memory consumptions of trees, tries and multibit tries vary case by case. TC-trie is a dynamic mixture of trees, tries and multibit tries. Suppose a large 32-bit IP address space in your mind; in each part of this space, the TC-trie acts in a way that the data structure reaches the maximum compression ratio. It means that in a situation like Fig. 4-A, the TC-trie would be a tree and in situations like Fig. 4-B and 4-C, the TC-tire would be a trie and a multibit trie respectively. Table 1 summarizes the results of the speed and memory-consumption comparison between trees, binary tries, and multibit tries of Fig. 4.

For a moment, forget combination of trees with tries and just consider the combination of tries with different multibit tries. This structure is a variable stride multibit trie. A variable stride multibit trie is a multibit trie that the number of strides and length of each stride vary in different paths from the root to the nodes. It could be said that the TC-trie is a variable stride multibit trie that its sparse parts are dynamically represented by binary sorted tree structures.

E. Related Works

Several IP address lookup solutions have been proposed previously which are based on tries. Some of them compress the trie for decreasing the memory consumption and improving the speed [1-2], [19-21]. Since a binary trie needs lots of comparisons and memory accesses, many solutions use multibit trie concept to accelerate the lookup search [3-13]; however, using a multi-bit trie instead of binary trie normally increases the memory consumption. Some papers have proposed compression methods for solving this problem [8]-[13].Compression methods usually suffer from update overhead. Besides methods that work based on trie, other methods are also proposed that some of them use hash tables [14] while some others use CAMs (Content Addressable Memories) or TCAMs (Tertiary CAMs) for solving the LPM problem [15-17]. Some of these methods are not fast enough and some others consume lots of memory and those ones which have good speed and memory consumption suffer from heavy update overhead. A comparison between some of these methods comes in [18]. It seems that finding a method that meets all of the requirements of high-speed search, low memory consumption, and high-speed update is an endless challenge.

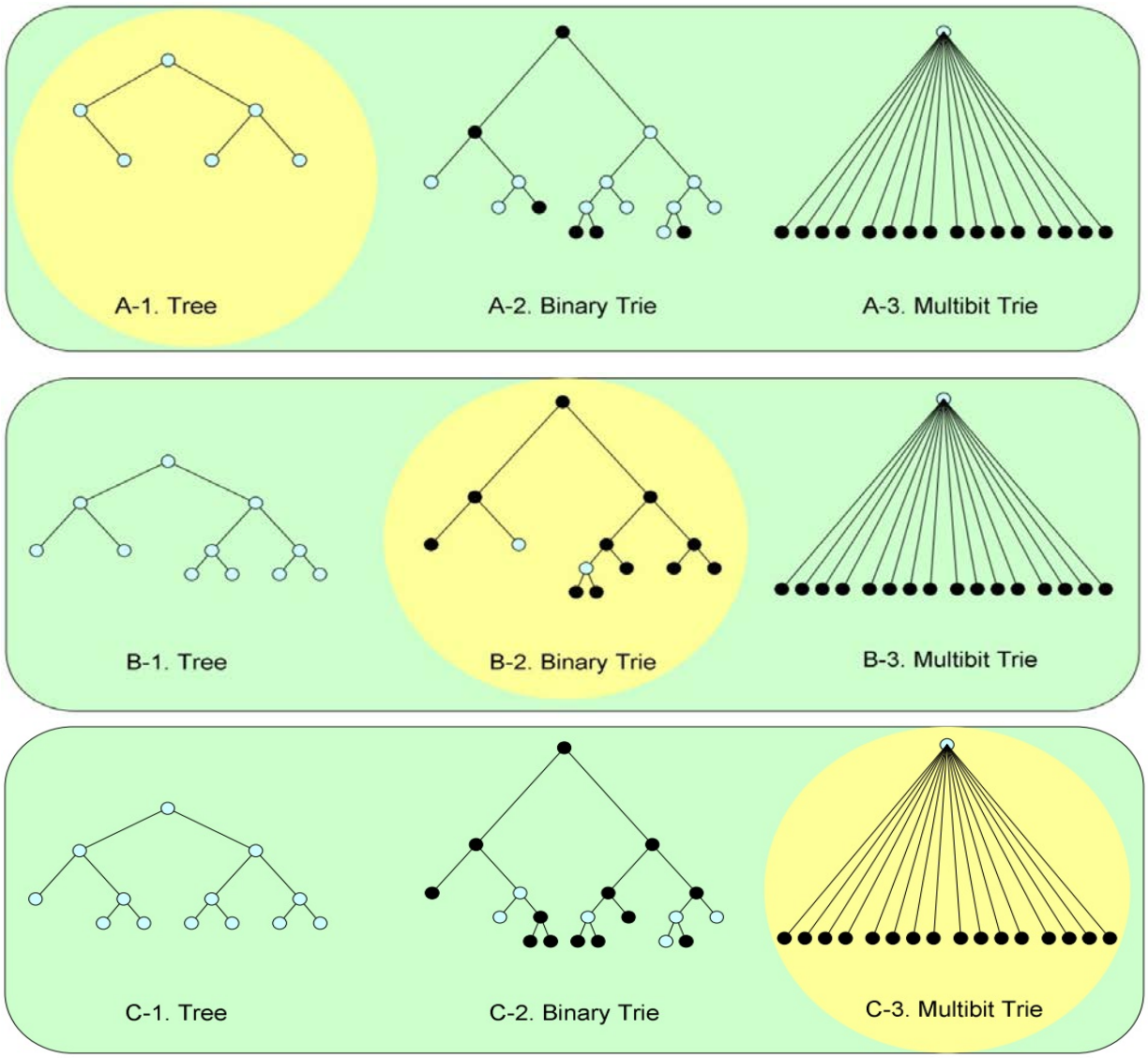


Fig. 4. Comparing the memory consumption of trees, binary tries and multibit tries in a 4-bit address space. A. Tree consumes less memory than trie and multibit trie. B. Binary trie consumes less memory than tree and multibit trie. C. Multibit trie consumes less memory than tree and binary trie

TABLE I. SPEED AND MEMORY CONSUMPTION COMPARISON BETWEEN TREES, BINARY TRIES AND MULTIBIT TRIES OF FIG. 4

		Number of Nodes	Size of Each Node (Byte)	Memory Consumption (Byte)	Maximum Depth
Example 1 (Fig. 4-A)	Tree	6	8	48	2
	Binary Trie	17	4	68	4
	MultibitTrie	16	4	64	1
Example 2 (Fig. 4-B)	Tree	11	8	88	3
	Binary Trie	13	4	52	4
	MultibitTrie	16	4	64	1
Example 3 (Fig. 4-C)	Tree	13	8	104	3
	Binary Trie	19	4	76	4
	MultibitTrie	16	4	64	1

II. TREE-COMBINED TRIE (TC-TRIE)

TC-trie is a flexible data structure that combines multibit tries with trees. Fig. 5 shows a binary trie representing the network prefixes in the 6-bit address space.We want to convert this structure to the TC-trie of Fig. 6. This conversion should be done in order to reduce the depth and memory consumption.

In our implementation for 32-bit IPv4 address space, the TC-trie starts with a stride of length 16-bit. Doing this, most significant 16 bits of address are considered at the first step and hence the depth of the structure never exceeds 17. Since 16 bits of each prefix are implicitly encoded in the first stride, all tree nodes should keep just the remaining 16 bits for each prefix. Therefore 16 bits from the prefix and 1 bit from the prefix length field will be saved.

A. Trie & Tree Nodes

For measuring the memory consumption, size of trie and tree nodes are required. Fig. 7 illustrates the trie and tree nodes. As this figure demonstrates a tree node in our architecture is exactly two times bigger than a trie node.

A trie node is composed of the following fields: Pointer: a pointer to a table in the next level that contains its children nodes. Len: a value between 0 and 15 that shows the length of stride minus one. For a binary node, len equals to 0 and for a node with degree 16 (a node at the head of a stride with length 4), len equals to 3.

Result: for the nodes that contain a valid prefix, result is a pointer to the NHT (Next-Hop Table); for other nodes, it has the reserved value of "11111111"$_2$ that means no network prefix exists for this node.

The following fields consist in a tree node: Trie Pointer / Result: if a tree node to be at the head of a multibit trie cone, this field is a pointer to a table in the next level that contains its trie children. Otherwise, eight least significant bits of this field compose a pointer to the NHT (Next-Hop Table) while all the other bits are set to one. Len: if a tree node to be at the head of a multibit trie cone, this field shows the length of stride minus one. Tree Pointer: a pointer to a two-entry table in the next level that contains its tree children nodes. Prefix: the remaining least significant16 bits of the prefix that corresponds to the tree node. Plen: length of the remaining part of the prefix minus one.

The following sub-sections explain how to build TC-trie from a binary trie and how to search it. The explanations about the incremental update and IPv6 implementation are ignored due to the lack of space.

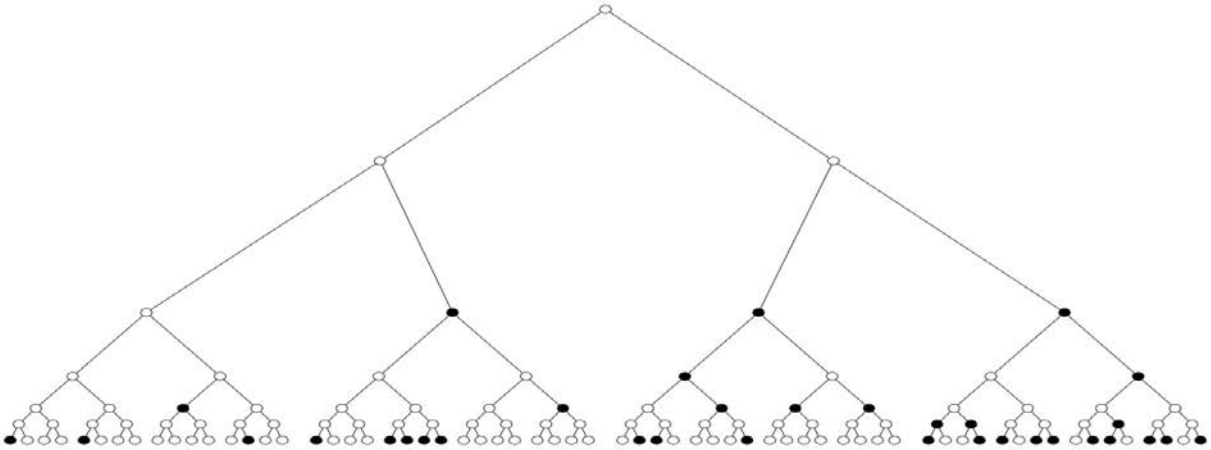


Fig. 5. A binary trie representing the network prefixes in a 6-bit address space

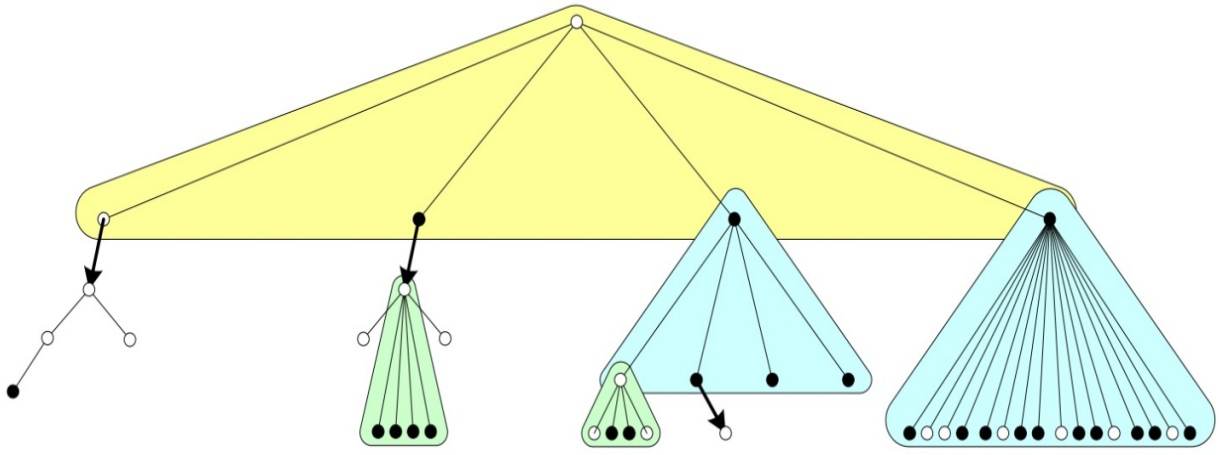


Fig. 6. A TC-trie equivalent to the binary trie of Fig. 5

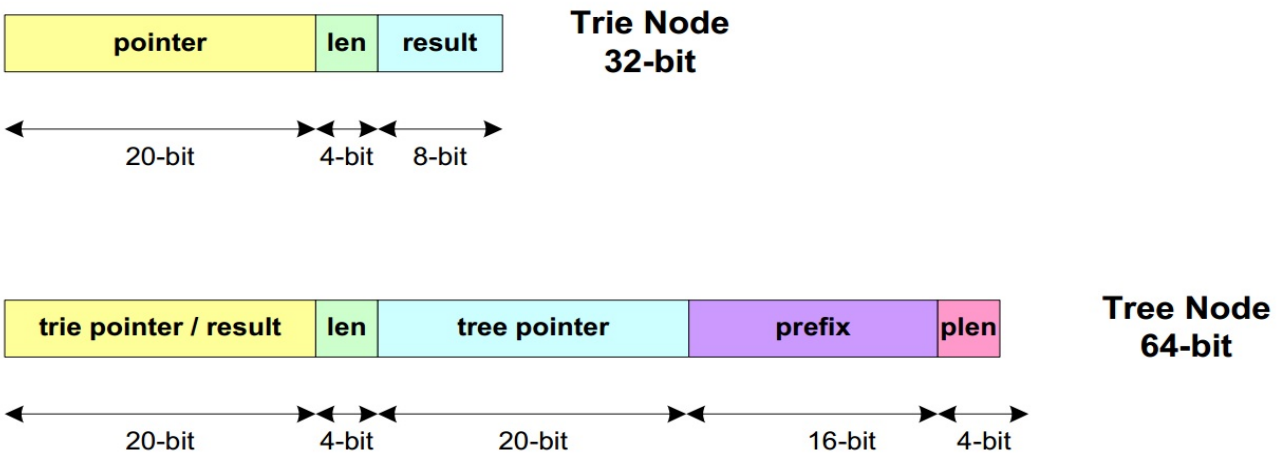


Fig. 7. Trie and tree nodes

B. Building TC-trie

For building a TC-trie, the starting point is a binary trie which is kept in the control unit of the router. The control unit (sometimes called slow path) of a router usually is a GPP (General Purpose Processor) that runs a routing protocol like RIP, OSPF or BGP. The control unit is responsible for updating the forwarding table of the lookup engine. Since CPE (Controlled Prefix Expansion) [6] in multibit tries and many compression techniques used in other IP lookup methods removes parts of information, the control unit must have an original copy of information inside itself to do the update operation properly. The control unit uses DRAM and update doesn't occur very frequently, so the size and speed of the original structure in the control unit is not critical. In our implementation, the control unit keeps a binary trie that contains the original non-scratched information. For building a TC-trie at the first time or for incrementally updating it, the control unit uses its binary trie structure.

To build a TC-trie from a binary trie, two main steps should be followed. The first step is finding dense regions and representing them with multibit tries. In the second step, prefixes which are not covered by multibit tries must be represented by binary sorted trees. When searching for dense regions, two issues should be considered. The first issue is the search resolution. Resolution equal to one means the maximum resolution that yields the most accurate results. Resolution equal to two means that searching the binary trie is being done with step size two. So the resulting multibit tries would be of depth 2, 4, 6 and etc. In general, if resolution equals to r, the depth of all multibit tries which are obtained would be a multiple of r.

The second issue is the threshold between denseness and sparseness concepts. How many prefixes have to be in a region of a trie to call it a dense region? To answer this question both memory consumption and lookup speed should be considered. Suppose that the resolution is four and we want to find out whether a cone with depth four in the original binary trie has the essential condition for being a stride of depth four or not. A stride of depth four needs 16 trie nodes that consumes 16*4=64 bytes of memory. On the other hand, 64 bytes is equal to 8 tree nodes. So, if the number of prefixes is less than eight, tree representation is more compact; otherwise, a stride of depth four is better. Therefore, by considering only the memory constraint,it could be said that a binary trie of depth d is dense if it contains at least 2 d-1 prefixes. We refer to this threshold as 50% threshold.

It's clear that a single stride is faster than any tree structures. So, if speed to be considered in addition to the memory consumption, the threshold should be less than 50%. Since compressing the higher parts of the trie improves the lookup speed of more prefixes, it's wise to apply different threshold for different heights of the trie. In other words, it's reasonable to increase the threshold from top to the bottom of the trie up to 50%.

C. Lookup Search

Fig. 8 illustrates an example of a lookup search on a TC-trie in a 6-bit memory space. This TC-trie is the one that was shown in Fig. 6. Notice how tree and trie nodes filled the memory space. Each word of the memory can be filled with one tree node or two trie nodes. Since each stride of multibit trie always has even number of nodes, no part of the memory space would be dissipated.

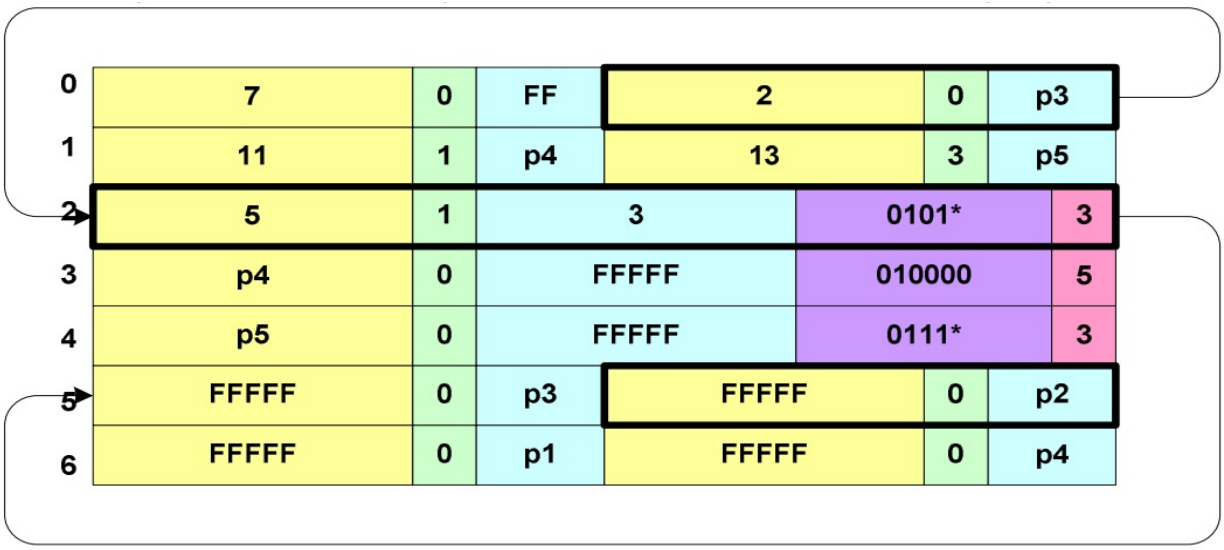


Fig. 8. An example of a lookup search in the memory architecture of a TC-trie structure

TABLE II. MEMORY CONSUMPTION, AVERAGE DEPTH, AND MAXIMUM DEPTH OF FIVE DIFFERENT FORWARDING TABLES [1]. THESE RESULTS ARE OBTAINED WITH RESOLUTION 2 AND 50% THRESHOLD VALUE

Table Name	Table Size	Memory Consumption (Byte)	Maximum Depth	Average Depth
AS1221	**156535**	1303024	11	3.98
AS267	**134024**	1136144	8	3.50
AS286	**134236**	1137232	8	3.49
AS3333	**139033**	1170904	10	3.53
AS3549	**133869**	1135136	8	3.51

TABLE III. MEMORY CONSUMPTION, AVERAGE DEPTH, AND MAXIMUM DEPTH OF AS1221 [1] FOR RESOLUTIONS. THESE RESULTS ARE OBTAINED WITH 50% THRESHOLD VALUE

Resolution Step	Memory Consumption (Byte)	Maximum Depth	Average Depth
2	1303024	11	3.98
4	1357288	11	3.92
6	1392992	8	3.95

TABLE IV. MEMORY CONSUMPTION, AVERAGE DEPTH, AND MAXIMUM DEPTH OF AS1221[1] FOR DIFFERENT THRESHOLD VALUES. THESE RESULTS ARE OBTAINED WITH RESOLUTION 2

Threshold value	Memory Consumption (Byte)	Maximum Depth	Average Depth
50%	1303024	11	3.98
25%	1969456	11	3.27
Variable	1890496	12	3.32

In this example, the query address is "010101". Fig. 6 shows how trie and tree nodes must be traversed for searching this address. In Fig. 8, nodes that must be read are highlighted. The final result is the value p2.

III. EXPERIMENTAL RESULTS

Different experiments have been done based on different forwarding tables, different resolutions, and different threshold values. Table 2 shows the memory consumption, average depth, and maximum depth of TC-trie structure achieved for five forwarding tables. These forwarding tables are obtained via potaroo website [22].

Table 3 shows the effect of changing the resolution in the memory consumption and depth of the forwarding table [1] BGP routing table analysis reports: http://bgp.potaroo.net/, retrieved on January 2000 & December 2014. AS1221 [22].

Table 3 shows the effect of changing the resolution in the memory consumption and depth of the forwarding table AS1221 [22].

In Table 4, the effects of changing the tree-trie threshold

are illustrated. This table shows that higher threshold (up to 50 %) yields lower memory consumption, while lower threshold yields smaller depth. In this table the last row stands for a variable threshold. This variable threshold increases from the root of the trie (original binary trie) to the leaves. The variable threshold causes higher parts of the trie have more chance of being compressed.

IV. Conclusion and Future Works

A new data structure for fast and memory-efficient IP address lookup was presented. This structure, which is a variable stride multibit trie combined with binary sorted tree was called Tree-Combined Trie (TC-trie).TC-trie collects benefits of multibit tries and binary sorted trees in itself. Dynamic reconfigurability of TC-trie made it a very flexible data structure that is scalable to the number of prefix, prefix distribution and prefix length. The proposed data structure prepares a smooth transition from IPv4 toward IPv6. Different aspects of this new data structure were considered and the building procedure and lookup search in this structure were explained. Examples and experiments demonstrated that our method consumes less memory than trie-based methods and is faster than tree-based methods. The flexibility of TC-trie is more than other methods and it better fulfills the requirements of current unsteady Internet. Our future work can be outlines as follows:

- Finding a better tree structure for combining with multibit tries by considering the following issues:

i. Multiway trees

ii. The sorting mechanism of the tree nodes

- Doing more theoretical and experimental studies about a threshold point between trees and tries.

i. Best static threshold conditions regarding the memory consumption and speed

ii. Dynamic threshold conditions

- Adding more intelligent to the system during the TC-trie build up.

i. How to assign priorities to the tree nodes to sort them in a way that the average TC-trie depth to be minimum.

- Simulating a scenario of growing IPv6 tables in an actual condition that may occur in the future.

Acknowledgments

This work has been supported by the Saeed Shamshiri.

References

[1] D. R. Morrison, “PATRICIA - Practical algorithm to retrieve information coded in alphanumeric,” J. ACM, vol. 15, no. 4, pp. 514–34, Oct. 1968.

[2] K. Sklower, “A tree–based packet routing table for Berkeley UNIX,” Proc.1991 Winter Usenix Conf., pp. 93–99, 1991.

[3] P. Gupta, S. Lin, and N. McKeown, “Routing lookups in hardware at memory access speeds,” Proc. IEEE INFOCOM ’98, pp. 1240–47, Apr. 1998.

[4] Tomas Henriksson, Ingrid Verbauwhede, “Fast IP address lookup engine for SOC integration,” Proc. of Design and Diganostics of Electronic Cricuits and Systems, Brno, Czeck Republic, pp. 200-210, Apr 2002.

[5] Chen, W.E.; Tsai, C.J. “A fast and scalable IP lookup scheme for high-speed networks,” Proc. IEEE ICON99, pp. 211-218, 1999.

[6] V. Srinivasan and G. Varghese. “Fast address lookups using controlled prefix expansion,” ACM Transactions on Computer Systems, vol. 17, no. 1, pp. 1-40, Feb. 1999.

[7] T. Chiueh and P. Pradhan, “High performance IP routing table lookup using CPU caching,” Proc. IEEE INFOCOM'99, New York, NY, USA, pp. 1421-1428, April 1999.

[8] KARI SEPPANEN, “Novel IP address lookup algorithm for inexpensive hardware implementation”, WSEAS Transactions on Communications, vol. 1, no. 1, pp. 76-84, 2002.

[9] Nen-Fu Huang, Shi-Ming Zhao, Jen-Yi Pan, and Chi-An Su, “A fast IP routing lookup scheme for gigabit switching routers”, Proc. IEEE INFOCOM, pp. 1429-1436, Mar. 1999.

[10] Stefan Nilsson, Gunnar Karlsson, “Fast address lookup for internet routers”, Proc. IFIP 4th International Conference on Broadband Communications, pp. 11-22, 1998.

[11] S. Nilsson and G. Karlsson “IP-address lookup using LC-tries,” IEEE JSAC, vol. 17, no. 6, pp. 1083–92, June 1999.

[12] M. DegerMark, et al., “Small forwarding tables for fast routing lookups,” Proc. ACM SIGCOMM 97, pp. 3-14, 1997.

[13] Derek Pao, Cutson Liu, Angus Wu, Lawrence Yeung and K. S. Chan, “Efficient hardware architecture for fast IP address lookup,” IEE Proceedings-Computers and Digital Techniques, vol. 150, no. 1, pp. 43-52, Jan. 2003.

[14] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, “Scalable high speed IP routing lookups”, Proc. ACM SIGCOMM ’97, pp. 25–36, Sept. 1997.

[15] Huan Liu, “Routing table compaction in ternary CAM”, IEEE Micro, vol. 22, no. 1, pp.58-64, January 2002.

[16] Francis Zane, Girija Narlikar, Anindya Basu, “CoolCAMs: power-efficient TCAMs for forwarding engines”, IEEE INFOCOM, vol. 1, pp. 42-52, 2003.

[17] Anthony J. McAuley, Paul Francis, “Fast routing table lookup using CAMs”, Proc. IEEE INFOCOM, pp. 1382-1391, March/April 1993.

[18] Miguel Á. Ruiz-Sánchez, Ernst W. Biersack and Walid Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Network Magazine, vol. 15 no. 2, pp. 8-23, March/April 2001.

[19] Yi-Mao Hsiao , Yuan-Sun Chu, Jeng-Farn Lee, Jinn-Shyan Wang, “A high-throughput and high-capacity IPv6 routing lookup system” , Computer Networks, Elsevier, 2013.

[20] KunHuang , GaogangXie , YanbiaoLi , DafangZhang, “Memory-efficient IP lookup using trie merging for scalable virtual routers”, Computer Networks, Elsevier, 2014.

[21] Hyuntae Park, Hyejeong Hong, Sungho Kang, “An efficient IP address lookup algorithm based on a small balanced tree using entry reduction”, Computer Networks, Elsevier, 2012.