IP Lookup Technology for Internet Computing

Yen-Heng Lin and Sun-Yuan Hsieh(✉)

Department of Computer Science and Information Engineering,
National Cheng Kung University, Tainan, Taiwan
hsiehsy@ncku.edu.tw

Abstract. Internet protocol (IP) lookup is a key technology that affects network performance. Numerous studies have inves- tigated IP lookup and provided solutions for improving lookup algorithms. Herein, we utilized various state-of-the-art algorithms for improving IP lookup performance and explore trie-based algorithms to understand how these algorithms affect memory access or usage and the resulting reductions in IP lookup times. Moreover, we utizsed parallel data processing for increasing IP lookup throughput. These algorithms are applicable to all tries. Nevertheless, we conducted experiments by using only binary tries for simplicity. IP lookup algorithms were tested through simulations using real IPv4 router tables with 855,997 or 876,489 active prefixes. Finally, a synthetic architecture combining all of the discussed algorithms was proposed and evaluated.

Keywords: Internet computing · Router table design · IPv4 · Trie-based algorithms · Direct lookup tables · Next-hop tables · Data communication

1 Introduction

With the rapid development of the Internet, services such as social media, streaming media, and cloud computing [1] have gradually become a part of daily life. High-speed Internet is necessary for handling large numbers of internet protocol (IP) packets; however, Internet speed is affected by IP lookup, a key technology. To send an IP packet to a destination address, it must be transmitted through multiple routers; nevertheless, connections between routers are not one-to-one links but are many-to-many links. If a router is to transmit an IP packet to another router, it uses an IP lookup algorithm to determine an appropriate next hop to the next router on its own routing table. This IP lookup process is the bottleneck in Internet communication. Moreover, if the transmitted IP packets are all of minimal size (40 bytes) and if the aggregated link bandwidth is 40 gigabits per second (Gbps), then IP lookup should be performed 125 million times per second (125 million lookups per second, Mlps) to meet the required throughput of 8 ns per lookup in this worst-case scenario.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2023
Published by Springer Nature Switzerland AG 2023. All Rights Reserved
Z. J. Haas et al. (Eds.): WiCON 2022, LNICST 464, pp. 156–175, 2023.
https://doi.org/10.1007/978-3-031-27041-3_12

Classless Inter-Domain Routing (CIDR) [2] was introduced in 1993 as a replacement for the previous classful network addressing architecture on the Internet. Routing table prefixes no longer have fixed lengths of 8, 16, or 24 bits for 32-bit IPv4 [3] addresses; instead, their lengths may vary. In CIDR, a prefix is formatted as a pair of an address and prefix length, as presented in Fig. 1(a); in this figure, the leftmost prefix length address bits indicate the prefix, and the remaining bits can be ignored. For example, the two-bit entry (10*/2 C) indicates that the prefix is 10_2 and the corresponding next hop is router C. When a router receives a incoming packet, it extracts the destination address of the packet and compares it with each prefix entry in the routing table, starting from the leftmost significant bit of the destination address. If numerous prefixes match the address, the router chooses the prefix with the longest length as the best matching prefix (BMP) and forwards the IP packet to the corresponding next-hop router with the longest prefix match (LPM). This is the IP lookup process. For example, in Fig. 1, router A receives an IP packet with a 5-bit destination IP address "10100" that matches the prefixes 1 and 10; therefore, the router chooses 10 with prefix length 2 as the BMP and forwards the packet to the corresponding next-hop router C.

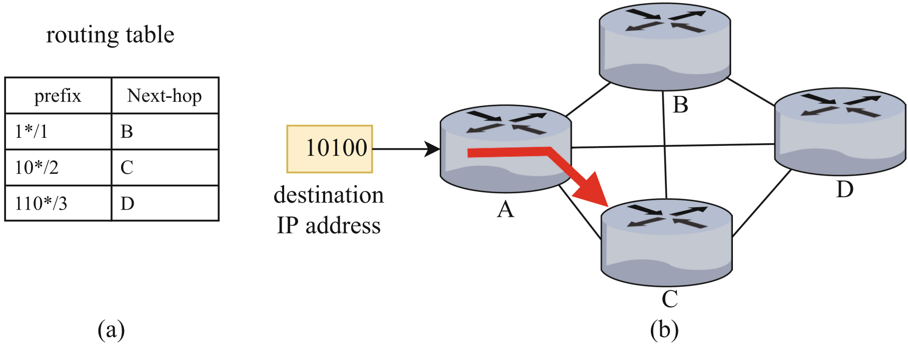


Fig. 1. (a) Routing table of router A. (b) Forwarding of an incoming IP packet with a 5-bit IP address

Over the past few decades, numerous studies have investigated implementations of IP lookup methods. These methods can be classified into two major categories: hardware-based and software-based methods. In hardware-based methods, parallel data processing can be used to increase lookup speed. Ternary content addressable memory is a prominent hardware-based method [4] and supports highly parallel lookups for entire entries; because it returns the BMP within a single memory access time, its lookup rate can exceed 250 Mlps. However, this method has high power consumption and requires substantial physical space, reducing its practical applicability. In addition, owing to the development of Network Function Virtualization (NFV) technology [5], software-based methods are more suitable than hardware-based methods for applications that require

portability. Software-based methods can be further divided into range-based, hash-based, and trie-based methods.

For range-based methods, Zec et al. proposed DXR [6] as an attempt to transform large routing tables into compact lookup structures (range tables) that easily fit into the cache hierarchies of modern CPUs. They further included an additional direct lookup table to divide the range table into 2^k chunks; in the IP lookup process, the first k bits of the destination address are used to reach a certain chunk and identify the LPM more quickly in a smaller range table.

Hash-based IP lookup methods have also been successful. For example, [7] revealed that hashing is an efficient method of achieving an exact match but not LPM. Because the prefixes in the hash table have arbitrary length, the number of leftmost significant bits that must be extracted from the destination address to perform a hash table lookup is unknown. To improve on hash-based methods, Dharmapurikar et al. [8] proposed a method that uses parallel Bloom filters to assess whether a prefix matches an address of a specific length, followed by quickly deleting addresses whose length is inconsistent with the prefix and may otherwise be extracted to the hash table for lookup. Fan et al. proposed another replacement method [9] in which the Bloom filter is replaced with a cuckoo filter to reduce false positives and improve the lookup performance.

Finally, trie-based methods use trie structures to implement IP lookup. The basic trie-based method is the binary trie [10]; however, a drawback of this method is that the tree must be traversed from the root to a leaf to determine the LPM. To alleviate this disadvantage, the priority trie [11] was developed, which uses priority marks to determine whether any longer prefix exists in the child nodes to avoid wasting lookup time. In the fixed-stride multibit trie [12], 2^k children of a node are used to seek k bits at a time, reducing the time complexity from $O(W)$ (binary trie, W = the length of the IP address) to $O(W/k)$. Moreover, the LC-trie [13], patricia trie [14], poptrie [15], and CP-trie [16] have been proposed to increase lookup rate by reducing the trie height and memory consumption.

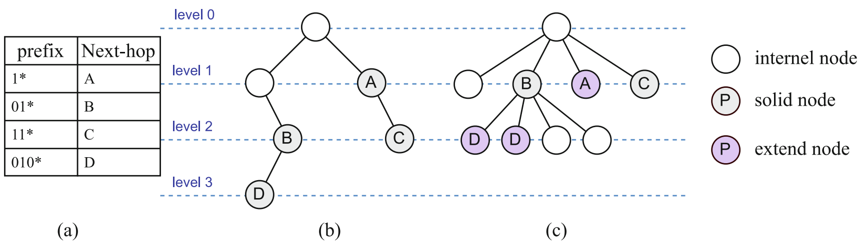


Fig. 2. (a) Routing table. (b) Corresponding binary trie. (c) Corresponding 2-bit stride multibit trie.

In this paper, we focus on trie-based methods with a simple, tiny data structure. Several relevant studies have proposed improvements to the performance of trie-based lookups. Accordingly, we attempt to unify commonly used technologies to help readers understand how this technology improves IP lookup performance. To clearly explain each technology, we include examples with the basic binary trie and the corresponding algorithms for each technology to facilitate the understanding and implementation of these technologies. Moreover, five metrics were used to measure the performance of each technology in our experiment: average Mlps, memory consumption, trie height, average memory accesses, and average update time. These metrics were compared to understand whether the various technologies perform as described.

The remainder of this paper is organized as follows: Sect. 2 describes the binary trie scheme and the k-bit multibit trie scheme and explains how data are stored and searched in trie-based methods. Section 3 describes a direct lookup table algorithm that reduces memory access in the trie. Section 4 describes methods of branching the search of the direct lookup table and trie. Section 5 describes a next-hop table algorithm that reduces the memory consumption of the trie. Section 6 proposes a novel method of synthesizing previous technologies in a trie-based method. Section 7 describes a parallel data processing method that uses multiple threads to increase IP lookup throughput. Section 8 presents the experimental results for a comparison of each algorithm on real IPv4 tables with 855,997 and 876,489 active prefixes and presents the performance of the synthetic method. Finally, Sect. 9 provides our conclusion.

2 Trie-based Scheme

A binary trie is the basic trie-based method. Each node has at most two children, and prefixes can be organized on a digital basis. The search begins from the root, and the bits of the prefix are used to direct the branching. For a bit value of 0, the search moves to the left branch; for a bit value of 1, it moves to the right branch. The search continues until all bits of the prefix are evaluated, and the corresponding next hop is stored in the final branch node. The node storing the next hop is called a solid node; other nodes are called internal nodes. That is, the path from the root to the solid node is equivalent to the prefix, and the value in the solid node is the next hop corresponding to that prefix. Figure 2 presents a routing table stored in a binary trie. For example, the router (010*/3, C) is stored beginning at the root; the first bit is 1, meaning that the search proceeds to the right branch, and the second bit is 1, meaning that the search proceeds to the right branch. Finally, the next hop C is stored in the node.

Searching for the LPM of the destination address on a binary trie is a top-down process from the root. One bit of the destination address is examined at a time. If the bit value is 0, the search proceeds to a left child; otherwise, it proceeds to a right child until it finally proceeds to a null child. When the search reaches a solid node, all branches passed are recorded as the current LPM, and the value stored in the solid node is the corresponding next hop. When the

search has been completed, the next hop corresponding to the current LPM is returned. For example, to look up the 5-bit destination address 11011 in Fig. 2, the search begins at the root. The first bit is 1; thus, the search proceeds to right child. This child is a solid node; thus, the next hop A corresponding to LPM 1 is stored. The second bit is also 1; thus, the search again proceeds to the right child, which is also a solid node. We thus store the next hop C corresponding to LPM 11. Finally, the third bit is 0, and the left child is null; therefore, the search stops, and the next hop C corresponding to LPM 11 is returned. Figure 3 presents these search processes.

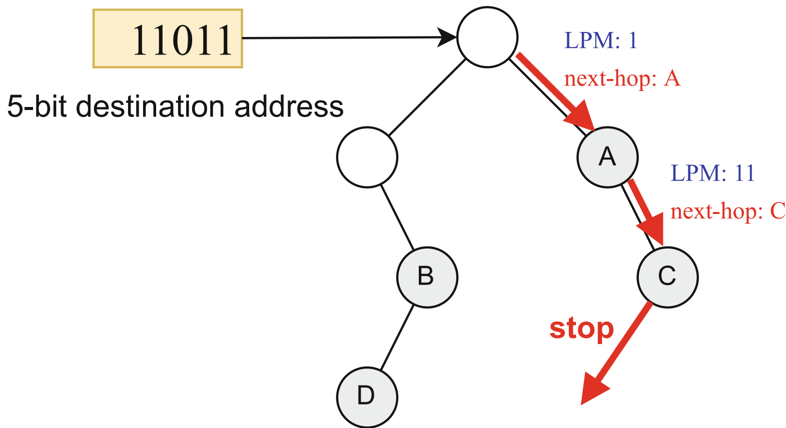


Fig. 3. Search steps for 5-bit destination address in binary trie

In contrast to the binary trie, in the k-bit stride multibit trie, each node has at most 2^k children, and prefixes are stored by using k bits of the prefix at a time to direct the branching. For the example in Fig. 2, to store a router (1*/1, A) in a 2-bit stride multibit trie, 2 bits are extracted from prefix 1. The length of the prefix is less than 2; thus, we must expand the prefix to obtain a set {10,11}. On the basis of this set, the next hop A is stored in both the third child and the fourth child from the left. To store another router (01*/2, B), the store operation directly proceeds to the second child and stores the next hop B in it. However, when the store operation attempts to store a third router (11*/2 C), the store proceeds to third child and determines that the next hop A is already stored in the node. The operation must determine whether the original stored value can be modified. The exact prefix length corresponding to this solid node is unknown because it may have been expanded from a prefix of insufficient length. Thus, more information must be stored in the multibit trie: the prefix length must be stored in the solid node as well. With the prefix length, a decision can be made

on whether to save the next hop C because the original next hop A has a prefix length of 1. After a comparison based on prefix length, the next hop A can be changed to C.

The search process for the LPM in a multibit trie is similar to that in a binary trie; however, k bits can be examined simultaneously. Therefore, the multibit trie may be up to k times faster than the original binary trie. In practical applications, the multibit trie does not achieve this performance because the k-bit stride multibit trie requires additional memory consumption due to prefix expansion, thus resulting in more cache misses. Figure 4 presents a search for the 5-bit destination address 11011 in the 2-bit stride multibit trie in Fig. 2. Regarding the binary trie, the next hop C corresponding to LPM 11 is returned.

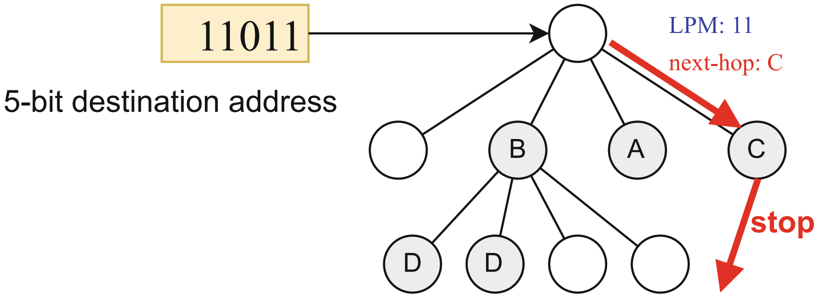


Fig. 4. Search steps for 5-bit destination address in 2-bit stride multibit trie.

3 Direct Lookup Table

Among IP lookup algorithms, trie-based algorithms tend to have lower memory requirements but require a longer search time for the LPM because internal nodes must be traversed to reach the leaf node, resulting in multiple expensive memory accesses. Previous studies [6,15,17] have used an additional array as a lookup table that can examine δ bits of the destination address in O(1) time to direct branching. In this strategy, instead of by traversing internal nodes in the trie, the corresponding next hop can be determined for most destination addresses by using this lookup table only; this can thus reduce the average number of memory accesses. Herein, this strategy is termed a direct lookup table.

The original concept of the direct lookup table was DIR-24-8 [17]; the strategy has two data structures: TBL24 (a direct lookup table) and TBLlong. TBL24 has 2^{24} elements, and each element has a size of 16 bits; one bit is used to determine whether the remaining 15 bits are the next hop or an index to TBLlong. To search in DIR-24-8, the most significant $\delta = 24$ bits of the destination address are first extracted as the index for TBL24, and the value of the element is evaluated. If the first bit of this value is 0, the remaining 15 bits are returned as the next hop; otherwise, the remaining 15 bits multiplied by 256 and then added to the remaining 8 bits of the destination address are used as the index for

TBLlong index. The value of the element in TBLlong is the next hop. Figure 5 presents the search process for a 32-bit destination address. For the incoming destination address (1.0.0.0), the search is directly indexed to 1.0.0 in TBL24, the first bit of which is 0; thus, the next hop A is returned. For the incoming destination address (1.0.1.1), the search is directly indexed to 1.0.1 in TBL24. In this case, the value of the first bit is 1; therefore, a lookup in TBLlong at the index $2 \times 256+1$ is performed, and the next hop C is returned.

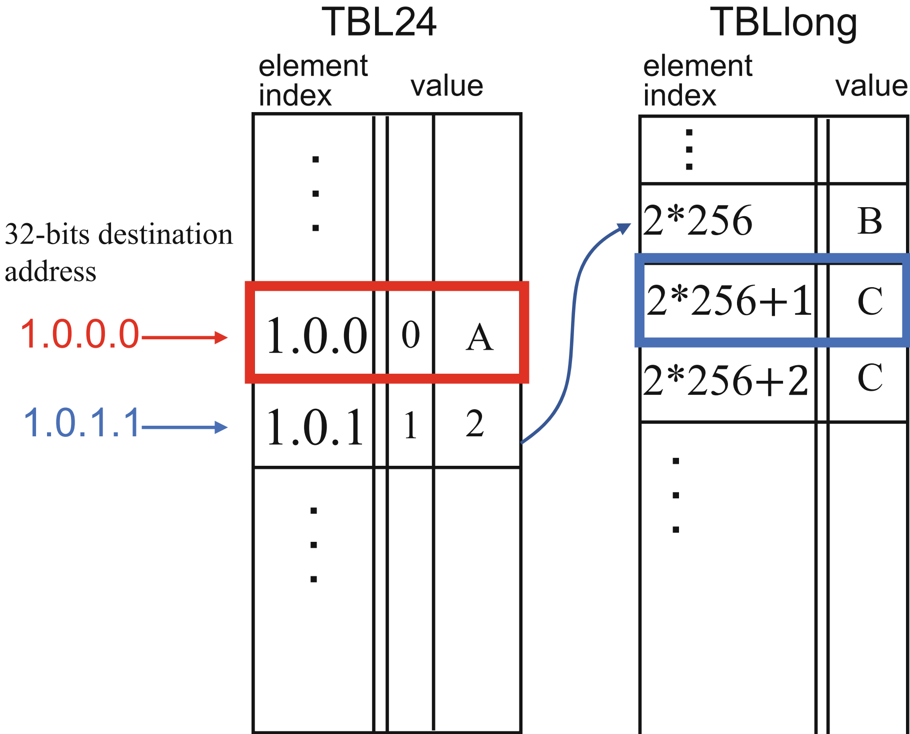


Fig. 5. Two searches with a 32-bit destination address in DIR-24-8 basic.

3.1 Level Pushing

A clear description of applying this concept to tries is presented in [16]. First, the trie corresponding to the router table is constructed. Second, nodes in level $< \delta$ are pushed to level δ, and the next hops stored by all nodes in level δ are stored in a direct lookup table with 2^{δ} elements. If the trie is searched for the next hop corresponding to an LPM before level $delta+1$, the direct lookup table is queried by extracting the most significant δ bits of the destination address as an index to obtain the result, reducing the number of memory accesses from the root to level δ. For example, Fig. 6(b) presents the binary trie corresponding

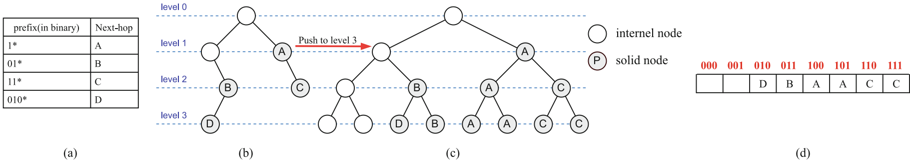


Fig. 6. (a) FIB table. (b) Corresponding Binary trie. (c) Push to level 3. (d) Corresponding direct lookup table for level 3.

to the forwarding information base (FIB) table in Fig. 6(a). To obtain a direct lookup table for $\delta = 3$, all nodes (both solid and internal) in levels 1 and 2 to are pushed to level 3. After level pushing, all node values in level 3 can be stored to the direct lookup array with 8 (2^3) elements from index 0 (000_2) to 7 (111_2). To look up the IP address 110*, the most significant $\delta = 3$ bits of the address are used to fetch index 6 ($= 110_2$), obtaining the next hop C.

3.2 Priority Table

To store a next hop with prefix length $\leq \delta$ in the direct lookup table, recursive level pushing from level 0 to level δ must be used. However, this process creates numerous unused internal nodes, thus wasting memory space. To avoid this situation, next hops can be directly stored in the direct lookup table by expanding prefixes, as described in [18]. For example, to store the prefix 1* in the direct lookup table in Fig. 6, the prefix 1* with length 1 must be expanded to a prefix with length 3; thus, the prefix 1* is left shifted by two bits, and all possible combinations of two bits ({00,01,10,11}) are added to it to obtain four corresponding prefixes: 100*, 101*, 110*, 111*. The corresponding next hop A is then stored in index 4 (100_2), 5 (101_2), 6 (110_2), and 7 (111_2). The process for prefix 01* is similar; however, when prefix 11* is stored in the direct lookup table, prefix 11* is expanded to obtain the corresponding indexes 6 (110_2) and 7 (111_2). The next hop C cannot be stored in these indexes directly because the the next hop A has already been stored in these indexes. The storage of the next hop A in index 6 (110_2) indicates that a corresponding prefix that may be 1*, 11*, or 110* exists. If the corresponding prefix length is ≤ 2, then the next hop A can be changed to C; otherwise, it cannot. Because information on only the next hop is stored in the direct lookup table, information on the original prefix length has been lost, and whether a next hop with an existing index should be stored cannot be determined. To store the corresponding prefix length for each index, a priority table for the direct lookup table can be included.

The priority table has the same number of elements as the direct lookup table. The initial value of the priority table is 0, and when a next hop is stored in an index (e.g., 6), the value in that index in the priority table is checked. If the prefix length is greater than or equal to the value in the priority table, the next hop can be stored to the direct lookup table, and the value in the priority table is changed to the prefix length. Otherwise, no change is made. Figure 7

presents the same level pushing steps as those in Fig. 6 but with a priority table. When the prefix 11* is expanded and collides with the prefixes 110* and 111*, the priority values can be checked. All are 1 (< 2); thus, the next hop is changed to C, and the priority value is changed to 2. The same steps are performed for the prefix 010*.

This storage method can be summarized into two cases as follows:

- Prefix length $l < \delta$: Expand the prefix length to δ and derive the prefix expansion set ($\{P_l \ll (\delta - l)$ to $(P_l + 1) \ll (\delta - l) - 1\}$). Use this set to assess the priority values in the corresponding index individually. If the priority value $\leq l$, store the indexed next hop in the direct lookup table and modify the priority value to l. Otherwise, take no action.
- Prefix length $l = \delta$: Store the next hop in the corresponding index in the direct lookup table. Modify the corresponding indexed priority value to δ in the priority table.

The level pushing algorithm for the priority table is presented in Algorithm 1.

Algorithm 1. LEVEL_PUSH(p, l, n, δ)
Input: Prefix p, length l, next-hop n, push to level δ
Output: Direct lookup table D
We initiate the direct lookup table D and priority table *priority* before performing this algorithm.

```
1 /* extend 0                                                      */
2 Start ← p ≪ (δ − l)
      /* extend 1                                                  */
3 End ← (p + 1) ≪ (δ − l) − 1
      /* Change the storage of the next hop depending on the priority
      value                                                        */
4 for i ← Start to End do
5     if l ≥ priority[i] then
6         D[i] ← n
          priority[i] ← l
```

4 C-index

When the direct lookup table is searched and the corresponding next hop is obtained, the next hop may not correspond to the LPM. It only corresponds to the LPM before the $\delta + 1$ level; thus, a trie search must still be performed to check whether longer prefixes exist after the level δ. If the absence of such matching prefixes in the trie can be ascertained, then the search can be avoided,

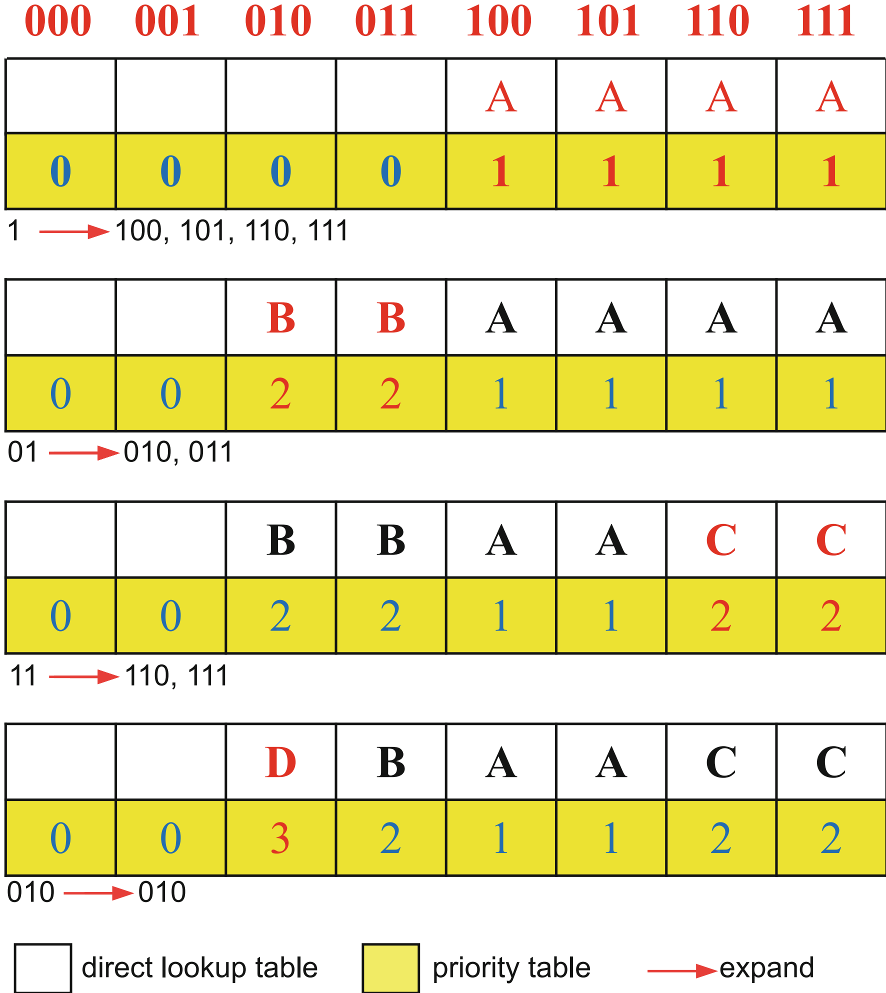


Fig. 7. Level push to level 3 with a priority table.

thus saving the trie lookup time. A study [16] proposed an additional flag table that indicates whether longer prefixes exist after the δ level; this table is called the C-index. The C-index table has the same number of elements as the direct lookup table. If a string longer than δ exists, the most significant δ bits of this prefix are extracted as the index to the C-index table, and the corresponding value in the C-index table is set to 1. For an incoming destination address, the most significant δ bits are used as the index for searching the direct lookup table and the C-index table. If the indexed value is 0, the trie search is unnecessary, and the result derived for the direct lookup table is returned. Otherwise, the trie search is performed to determine whether a longer LPM exists.

Figure 8 presents an example of a search process for a prefix with length $> \delta$ in a C-index table. First, the C-index values are initialized to 0. When the new router 0101*/4 D in Fig. 6 is added, the router information is stored in the corresponding binary trie. Because the prefix length $4 > \delta = 3$, the most significant 3 bits 010 are extracted as the index to the C-index, and the indexed value is modified to 1.

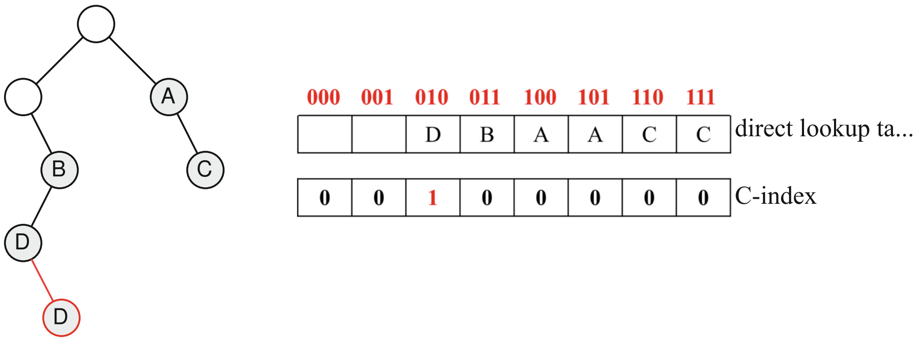


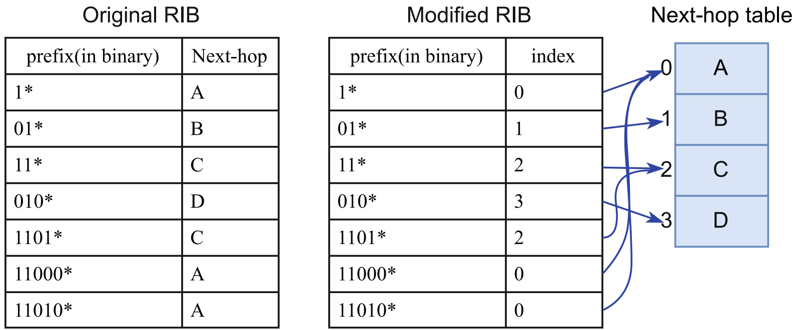
Fig. 8. Adding a new router 0101*/4 D to the trie in Fig. 6 in a C-index table.

5 Next-hop Table

The next-hop value in an IPv4 routing information base (RIB) table is a 32-bit value; thus, a 32-bit memory space is required to store each value. A 32-bit memory space can store 2^{32} different numbers; however, a RIB table typically stores far fewer next hops. In border gateway protocol (BGP) [19] snapshots, approximately 2^{10} different next hops are typically stored. Therefore, a next-hop table [6,15] with a 32-bit memory space for each element can be constructed to store the next hops, and a 10-bit memory space can be used to store the index of each element. The original RIB next-hop table of can be modified in accordance with this index to reduce the memory required to store the next hops.

For example, as indicated in Fig. 9, a 5-bit memory space is required to store each next hop in the original RIB table, and storing all seven routers requires 35 bits of memory. A next-hop table with four routers with 5-bit addresses requires 20 bits of memory space. The original RIB next-hop indexes are modified to the next-hop table indexes. Indexes 0 to 3 require only two bits of memory space; thus, storing all seven next-hop indexes requires only 14 bits of memory space. This strategy requires only 34 bits of total memory space, representing a 1-bit reduction.

We present an algorithm for next-hop table construction in Algorithm 2. The next hop n is used as the input. If n is not in the next-hop table, the next-hop table is resized, n is added to it, and the index is returned. Otherwise, the corresponding next-hop index is found and returned.



prefix(in binary)	Next-hop
1*	A
01*	B
11*	C
010*	D
1101*	C
11000*	A
11010*	A

prefix(in binary)	index
1*	0
01*	1
11*	2
010*	3
1101*	2
11000*	0
11010*	0

Store A~Z need 5-bit memory space for each next-hop.
And totall memory cost: 7*5= 35 bits

Store 0~3 need only 2-bit memory space for each index.
Store A~Z need 5-bit memory space for each next-hop.
And totall memory cost: 2*7+5*4=34 bits

Fig. 9. Reduction in memory consumption by using a next-hop table.

6 Synthetic Method

We also propose a synthetic binary trie method combining the aforementioned algorithms (direct lookup table, C-index, and next-hop table), and Fig. 10 presents its architecture. First, when an IP packet arrives, a next-hop table is used to convert the next hop to a next-hop index (Nidx). Second, the prefix length is detected, and if the length $l \leq \delta$, then the corresponding Nidx is stored in the direct lookup table. Finally, the remaining prefixes with length $l > \delta$ that

---

Algorithm 2. NEXTHOP_TABLE (n)
Input: Next-hop n
Output: Corresponding next-hop index idx
We initialize the next-hop table with N elements and size $size$ of 0 before performing this algorithm.

---

1 `/* Check all elements in the next-hop table and find the corresponding next-hop index. */`
2 **for** $i \leftarrow 0$ **to** $size - 1$ **do**
3   **if** $n = N[i]$ **then**
4 $idx \leftarrow i$
          return idx

5 `/* If n is not in next-hop table, add n to the table, resize the table, and return the corresponding index. */`
6 $N[size] \leftarrow n$
$idx \leftarrow size$
$size \leftarrow size + 1$
return idx

---

are not included in the direct lookup table have their corresponding Nidx stored in the binary trie, and the corresponding C-index value is modified.

Figure 11 presents a method of inserting a router in the synthetic architecture. For the incoming router 1*/1 A, the next hop A is set to Nidx 0, and the Nidx is stored in the expansion indexes 2 (10_2) and 3 (11_2). The value of the corresponding priority table is modified to prefix length 1. For a subsequent incoming router 10*/2 B, the next hop B is set to Nidx 1 and the Nidx is stored in the priority table. This next hop collides with the index for A in index 2 (10_2) of the direct lookup table. In the priority table, the priority value is $1 <$ prefix length 2; therefore, the indexed value is changed to Nidx 1 in the direct lookup table, and the indexed value in the priority table is changed to prefix

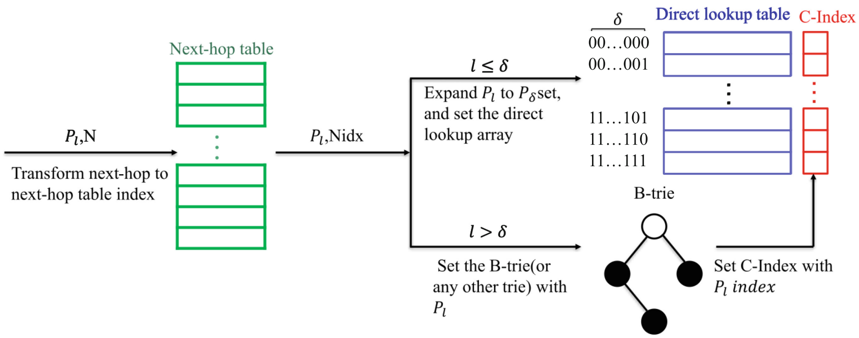
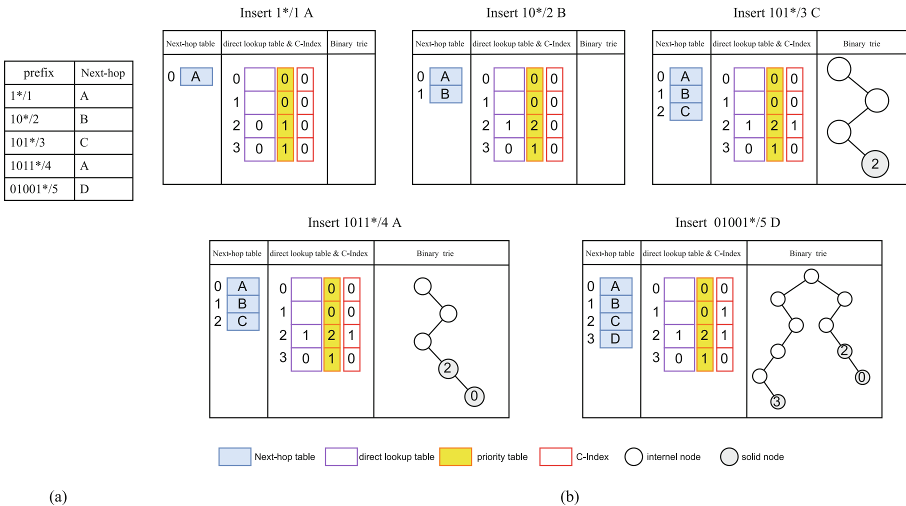


Fig. 10. Synthetic binary trie architecture.



prefix	Next-hop
1*/1	A
10*/2	B
101*/3	C
1011*/4	A
01001*/5	D

Fig. 11. (a) Router table. (b) Example of updating (inserting) routers in the synthetic architecture.

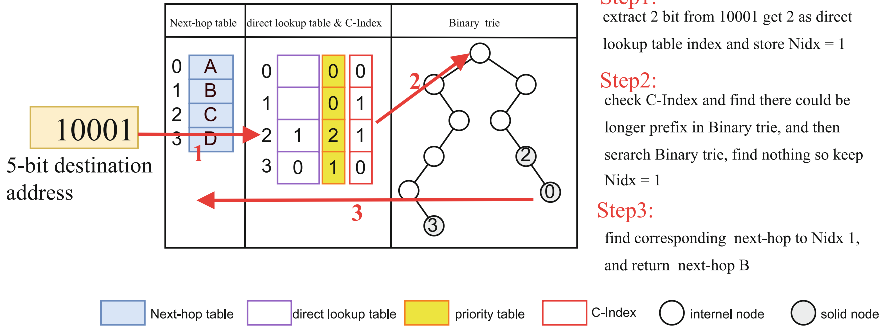


Fig. 12. Example of a lookup for a 5-bit destination address in the synthetic architecture.

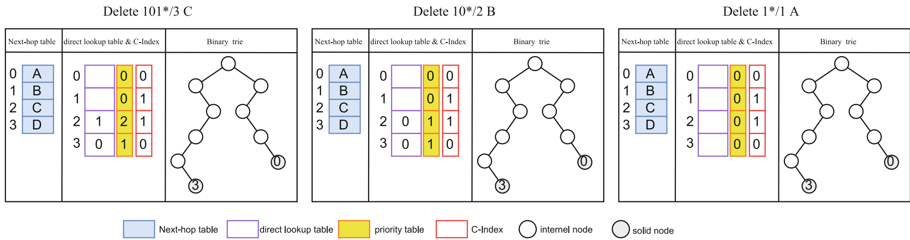


Fig. 13. Example of updating (deleting) a router in the synthetic architecture with $\delta = 2$.

length 2. For a third incoming router 101*/3 C, the next hop C is set to Nidx 2. Because the prefix length $3 > \delta = 2$, the Nidx is stored in the binary trie, and 2 bits of 101* are extracted. These bits indicate index 2 (10_2) of the direct lookup table, which already contains Nidx 1. Therefore, the indexed value in the C-index table is modified to 1. For a fourth incoming router 1011*/4 A, the next hop A is already set to Nidx 0. Because the prefix length $4 > \delta = 2$, Nidx 0 is also stored in the binary trie, and the first 2 bits of 1011* are extracted as index 2 (10_2) in the C-index, which is again set to 1. For a fifth incoming router 01001*/5 D, the next hop D is set to Nidx 3. Because the prefix length $5 > \delta = 2$, Nidx 3 is stored in the binary trie. Two bits of 01001* are extracted, indicating index 1 (01_2) in the C-index; the value in this index is set to 1.

Figure 12 presents a search for an incoming 5-bit destination address 10001. The first 2 bits of 10001 are extracted as 10; the value 2 (10_2) is used as the priority table index to obtain Nidx 1. However, the C-index value in this index is 1, indicating that another suitable Nidx corresponding to a longer matched prefix may exist in the binary trie. However, the binary trie search returns no matches. Finally, Nidx 1 is determined to be the LPM and is used to look up the next hop, which is B in the next-hop table.

Figure 13 presents the deletion of a router in the synthetic method. The deletion rules are similar to insert rules. If router 101*/3 C is to be deleted, the

prefix length 3> $\delta = 2$. Therefore, the stored information in the binary trie can be simply modified by changing the corresponding solid node to an internal node. However, the value of the C-index cannot be modified because another prefix may correspond to this index in the C-index. Second, if router $10^*/2$ B is to be deleted, the prefix length $2 \leq \delta$. Thus, the priority table and C-index value should be modified to contain only $1^*/1$ A. However, the router information is not stored, and the previous information is overwritten after the insertion of $10^*/2$ B. Therefore, the delete operation in the direct lookup table requires storing all routers in the binary trie. By referencing the trie, we can change the indexed value to 0 in the direct lookup table and to 1 in the C-index. Third, deleting router $1^*/1$ A is similar to the deletion of $10^*/2$ B. The binary trie is referenced, the indexed value is changed to the initial value in the direct lookup table, and the C-index value is set to 0.

7 Parallel Data Processing Method

For any software LPM algorithm, the lookup time of an IP packet cannot be reduced to less than one cycle time. The lookup time is limited by the speed of the processor; that is, the lookup throughput of a processor has an upper bound. To overcome this limitation, multiple processors can be used to simultaneously process multiple incoming IP packets.

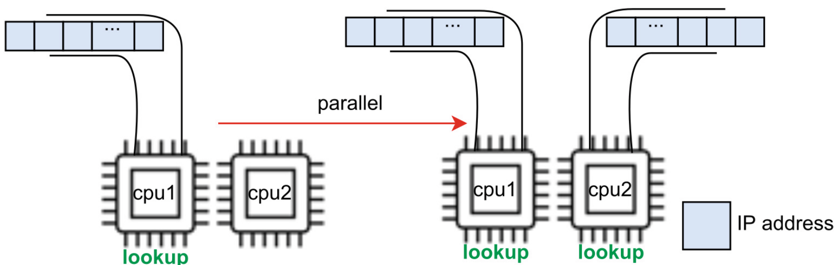


Fig. 14. Parallel LPM lookup with 2 CPU cores.

As illustrated in Fig. 14, only one processor initially performs LPM lookups for incoming IP packets. If two processors are used, the throughput can be duplicated if the execution time for each IP packet is constant. Increasing the number of processors should increase the theoretical throughput; however, memory resources still limit throughput in practice. If an excessive number of processors simultaneously run LPM lookup processes, a memory race condition is produced, which reduces throughput.

8 Experiment

We performed simulations using real data acquired from BGP router snapshots by using the C programming language on a computer with an AMD Ryzen 7

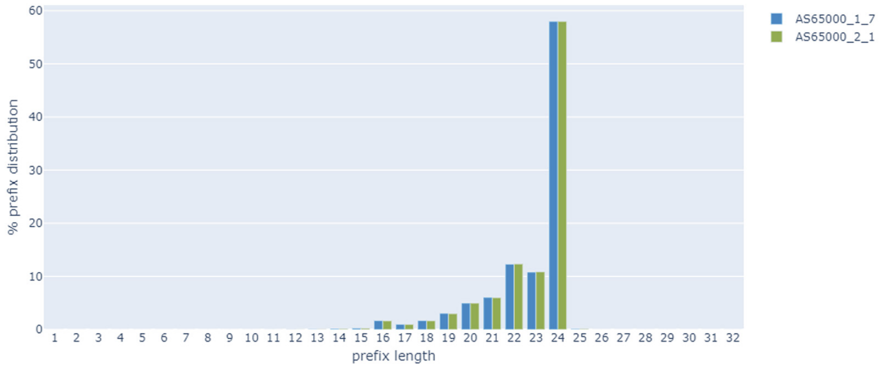


Fig. 15. IPv4 prefix length distribution.

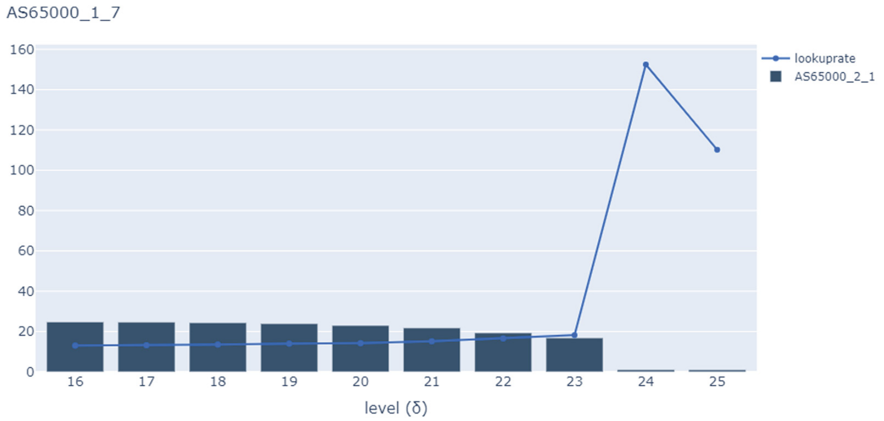


Fig. 16. Relationship between lookup rate and memory accesses on AS65000_1_7

4800H CPU with Radeon integrated graphics and 15.4 GB of memory. Two datasets were used, namely AS65000_1_7 and AS65000_2_1 with 855,997 and 876,489 active prefixes, respectively. Figure 15 presents the prefix length distribution for these two datasets.

We used five metrics to evaluate each algorithm: memory consumption, trie height, average lookup rate (Mlps), average memory accesses, and average update (insert/delete) time.

Table 1 presents the performance of the binary trie. Table 2 presents the performance of the binary trie with a direct lookup table and C-index. A comparison of the data in Tables 1 and 2 revealed that the inclusion of the direct lookup table could not reduce the trie height but could reduce memory access. According to Fig. 15, prefixes of length 24 constituted more than half of the router table. Therefore, a direct lookup table with $\delta = 24$ might minimize the average number of memory accesses and the maximum Mlps because querying the direct

lookup table in O(1) time without searching the trie is sufficient for half of all lookups in this case. As indicated in Table 2, the inclusion of a direct lookup table with a suitable δ could increase the search efficiency (Mlps) by a factor of 11.83. Figure 16 reveals that memory accesses are inversely proportional to Mlps.

Table 3 presents the performance of the binary trie with a next-hop table. Although the memory space was reduced to 3/4 of that used in the original binary trie, the number of memory accesses did not decrease; thus, the performance was not substantially affected and remained at approximately 13 Mlps. However, these experimental results do not indicate that a next-hop table is useless; this table is simply not suitable for use with an original binary trie. Because the binary trie requires little memory, reducing memory usage with a next-hop table is not effective. However, the inclusion of a next-hop table is useful for alleviating memory consumption in the synthetic architecture because the direct lookup table substantially increases memory consumption.

Table 4 presents the lookup performance corresponding to the architecture in Fig. 10. The performance was determined to be similar to that of the architecture presented in Table 2 for $\delta < 24$. Specifically, the memory space was reduced by approximately 25%. However, for $\delta = 24$, most of the data were stored in the direct lookup table, and the inclusion of the next-hop table helped reduce memory usage by approximately 50%, resulting in a 10% increase in overall performance compared with the architecture without a next-hop table.

Table 1. Evaluation of binary trie performance

AS65000_1_7					AS65000_2_1				
Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)	Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)
26.84	29	12.67	24.96	0.353	26.84	29	13.03	24.97	0.362

Table 2. Evaluation of binary trie performance with direct lookup and C-index tables for different δ values

δ	AS65000_1_7					AS65000_2_1				
	Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)	Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)
16	30.64	29	12.47	24.76	0.182	31.32	29	13.32	24.77	0.182
17	31.04	29	12.81	24.63	0.183	31.72	29	13.37	24.64	0.186
18	31.86	29	12.78	24.38	0.181	32.54	29	13.24	24.40	0.184
19	33.54	29	13.31	23.93	0.178	34.22	29	12.80	23.95	0.181
20	36.89	29	14.38	23.02	0.223	37.54	29	13.56	23.05	0.179
21	44.16	29	15.08	21.78	0.183	44.79	29	15.08	21.81	0.180
22	58.62	29	16.50	19.32	0.179	89.18	29	17.06	19.33	0.179
23	89.82	29	18.22	16.85	0.196	90.32	29	17.33	16.85	0.205
24	134.31	29	150.52	1.04	0.142	134.32	29	181.58	1.04	0.180
25	268.43	29	143.76	1	0.237	268.43	29	143.21	1	0.257

Table 3. Performance of a binary trie with a next-hop table

AS65000_1_7					AS65000_2_1				
Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)	Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)
20.13	29	12.70	24.96	0.353	20.13	29	13.05	24.97	0.362

Table 5 presents the lookup performance of the architecture displayed in Fig. 10 at $\delta = 24$ and n threads. As mentioned in Sect. 7, lookup throughput theoretically grows linearly with the number of processors. However, as listed in Table 5, increasing n from 1 to 2 increased the performance by a factor of only 1.8; doubling n to 4 increased the performance by a further factor of only 1.3. The improvement in theoretical throughput can be attributed to the memory race condition. However, parallel data processing is an effective option for increasing the throughput of an effective LPM algorithm. In our environment, parallel data processing achieved 501 Mlps on average at $n = 8$, reaching the standard speed of a wired connection.

Table 4. Performance of a binary trie with the synthetic method

δ	AS65000_1_7					AS65000_2_1				
	Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)	Memory consumption [MiB]	Trie height	Avg. Mlps	Avg. memory accesses	Avg. update time (s)
16	22.98	29	13.57	24.76	0.183	23.49	29	13.91	24.77	0.186
17	23.28	29	13.55	24.63	0.183	23.79	29	13.97	24.64	0.186
18	23.89	29	13.78	24.38	0.181	24.405	29	13.60	24.40	0.182
19	25.15	29	13.80	23.93	0.177	25.67	29	13.72	23.95	0.178
20	27.66	29	13.89	23.02	0.223	28.15	29	14.96	23.05	0.179
21	33.12	29	15.40	21.78	0.183	33.59	29	15.22	21.81	0.183
22	40.96	29	15.96	19.32	0.179	50.88	29	17.03	19.33	0.179
23	55.36	29	18.83	16.85	0.196	55.74	29	19.02	16.85	0.205
24	71.73	29	178.28	1.04	0.142	71.74	29	195.57	1.04	0.180
25	135.32	29	149.4	1	0.237	136.32	29	145.21	1	0.257

Table 5. Performance (Mlps) of a binary trie with level pushing to $\delta = 24$, a C-index, and a next-hop table with n threads (IPv4)

n	AS65000_1_7		AS65000_2_1	
	Avg.	Worst	Avg.	Worst
1	177.28	169.76	179.57	175.69
2	311.04	244.59	310.97	244.03
4	429.94	282.94	464.88	365.88
8	501.35	328.77	501.02	341.77
16	453.27	264.06	450.80	247.46

9 Conclusion

In this paper, we briefly introduce state-of-the-art lookup algorithms. We also conducted experiments to explore how they can increase the efficiency of IP lookup processes. State-of-the-art algorithms focus on reducing memory access and usage. The use of direct lookup tables can reduce memory accesses, which is the factor with the greatest effect on lookup performance. However, the effectiveness of this technology varies with the choice of δ. Because IPv4 router prefixes typically have a length of 24, we observed that setting $\delta = 24$ could produce a direct lookup table with the best performance; the average lookup rate exceeded 150 Mlps. In addition to trie memory accesses, reducing memory usage is critical. Our experimental results reveal that decreasing memory usage by half increased performance by 10%. Moreover, the average lookup rate exceeded 170 Mlps. These algorithms can be used to enhance the performance of binary trie methods, which initially could not achieve sufficient connection speeds. Furthermore, in addition to reducing the lookup time, parallel data processing can be used to increase the lookup throughput. In this study, the use of parallel data processing increased the average lookup rate to 450 Mlps, which is an excellent performance level.

References

1. Hayes, B.: Cloud computing. Commun. ACM **51**(7), 9–11 (2008). https://doi.org/10.1145/1364782.1364786
2. Fuller, T.V., Yu, J.L., Varadhan, K.: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, RFC 1519, Sept (1993)
3. Postel, J.: Internet protocol DARPA internet program protocol specification, RFC791, Sept. (1981)
4. Zheng, K., Hu, C., Lu, H., Liu, B.: A TCAM-based distributed parallel IP lookup scheme and performance analysis. IEEE/ACM Trans. Netw. **14**(4), 863–875 (2006)
5. Cui, C., Deng, H., Telekom, D., Michel, U., Damker, H.: Network functions virtualisation
6. Zec, M., Rizzo, L., Mikuc, M.: DXR: towards a billion routing lookups per second in software. ACM SIGCOMM Comput. Commun. Rev. **42**(5), 29–36 (2012)
7. Jain, R.: A comparison of hashing schemes for address lookup in computer networks. IEEE Trans. Commun. **40**(10), 1570–1573 (1992)
8. Dharmapurikar, S., Krishnamurthy, P., Taylor, D.E.: Longest prefix matching using bloom filters. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 201–212 (2003)
9. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than bloom. In: Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pp. 75–88 (2014)
10. Lim, H., Lee, N.: Survey and proposal on binary search algorithms for longest prefix match. IEEE Commun. Surv. Tutorials **14**(3), 681–697 (2011)
11. Lim, H., Yim, C., Swartzlander, E.E.: Priority tries for IP address lookup. IEEE Trans. Comput. **59**(6), 784–794 (2010)

12. Sahni, S., Kim, K.S.: Efficient construction of fixed-stride multibit tries for IP lookup. In: Proceedings Eighth IEEE Workshop on Future Trends of Distributed Computing Systems. FTDCS 2001, pp. 178–184 (2001)
13. Nilsson, S., Karlsson, G.: IP-address lookup using lC-tries. IEEE J. Selected Areas Commun. **17**(6), 1083–1092 (1999)
14. Sklower, K.: A tree-based packet routing table for Berkeley unix. USENIX Winter Citeseer **1991**, 93–99 (1991)
15. Asai, H., Ohara, Y.: Poptrie: a compressed trie with population count for fast and scalable software IP routing table lookup. ACM SIGCOMM Comput. Commun. Rev. **45**(4), 57–70 (2015)
16. Islam, M.I., Khan, J.I.: CP-TRIE: Cumulative popcount based trie for ipv6 routing table lookup in software and ASIC. In: Proceedings of the 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR), pp. 1–8. IEEE (2021)
17. Gupta, P., Lin, S., McKeown, N.: Routing lookups in hardware at memory access speeds. In: Proceedings of the IEEE INFOCOM 1998, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98, vol. 3, pp. 1240–1247. IEEE (1998)
18. Srinivasan, V., Varghese, G.: Fast address lookups using controlled prefix expansion. ACM Trans. Comput. Syst. **17**(1), 1–40 (1999). https://doi.org/10.1145/296502.296503
19. BGP. https://bgp.potaroo.net/