Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 588–605
Computer Networks
www.elsevier.com/locate/comnet

Fast binary and multiway prefix searches for packet forwarding

Yeim-Kuan Chang *

Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, ROC

Received 24 June 2005; received in revised form 24 January 2006; accepted 12 May 2006
Available online 21 June 2006

Responsible Editor: J.C. de Oliveira

Abstract

Backbone routers with tens-of-gigabits-per-second links are indispensable communication devices to deploy on the Internet. The IP lookup operation is the most critical task that must be improved in routers. In this paper, we first present a systematic method to compare prefixes of different lengths. The list of prefixes can then be sorted and stored in a sequential array, which is contrary to the linked lists used in most of trie-based structures. Next, fast binary and multiway prefix searches assisted by auxiliary prefixes are proposed. We also developed a 32-bit representation to encode the prefixes of different lengths. For the large routing tables currently available on the Internet, the proposed multiway prefix search can achieve the worst-case number of memory accesses of three and four if the sizes of the CPU cache lines are 64 bytes and 32 bytes, respectively. The IPv4 simulation results show that the proposed prefix searches outperform the existing IP lookup schemes in terms of lookup times and memory consumption. The simulations using IPv6 routing tables also show the performance advantages of the proposed binary prefix searches. We also analyze the performance of the existing lookup schemes by concurrently considering the lookup speed, the update speed, and the memory consumption. Although the update speed of the proposed prefix search is worse than the dynamic routing table schemes with $\log(N)$ complexity for a table of N prefixes, our analysis shows that the overall performance of the proposed binary prefix search outperforms all the existing schemes.

Keywords: Binary search; CPU caches; Longest prefix match; Routing table

1. Introduction

Internet traffic continues to grow at an unprecedented rate due to the advent of the World Wide Web. This has put tremendous pressure on Internet providers who have to set up the necessary infrastructure to support this growth. A crucial part of this infrastructure is the router. Backbone routers with a link speed at several 10-gigabits-per-second (Gbps), such as OC-192, 10 Gigabits and OC-768, 40 Gigabits, are commonly deployed. These backbone routers have to forward millions of packets per second at each port. All tasks that have to be executed by the router after receiving a packet can be divided into time-critical (fast path) and non-time-critical (slow path) operations depending on the packet type and its frequency. Time-critical operations that are operated on the majority of the packets must be implemented in a highly

* Tel.: +886 6 2757575.
E-mail address: ykchang@mail.ncku.edu.tw

1389-1286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2006.05.005

efficient and optimized manner to keep up with the high link speed and router bandwidth. As described in "Requirements for IP Version 4 Routers" [26], this includes IP packet validation, packet lifetime control, checksum recalculation, destination address parsing, and IP table lookups. Among all the tasks performed by routers, the IP table lookups entail the most time-consuming process in which the destination addresses are looked up against a *forwarding table* by a *forwarding engine* that determines the next-hops in the network, where the packets should be sent.

The forwarding table must maintain an entry for every allocated network address block that is represented as a *route prefix*. However, the continuous growth of the number of hosts and networks has made the forwarding tables in the backbone routers grow very rapidly. To efficiently use the IP address space and slow down the growth of the forwarding tables, the classless IP subnet scheme called Classless Inter-Domain Routing (CIDR) was introduced. While CIDR reduces the size of the forwarding tables, the table lookup problem now becomes more complex. With CIDR, each route prefix in the forwarding table can vary from 1 to 32 bits instead of 8, 16, or 24 bits in Classful Address scheme. As a result, the search in a forwarding table can no longer be performed by exact matching because the length of the prefix cannot be derived from the address itself. The table lookup problem becomes the "Longest Prefix Match" problem, which determines the longest route prefix to a destination address because there may be more prefixes that match the destination address.

Router designers were challenged to come up with fast and efficient algorithms for solving the Longest Prefix Match (LPM) problem. Three metrics are primarily considered in designing routers:

1. *Search time*. This is the time taken by the forwarding engine to look up the forwarding table for the destination address of an incoming packet.
2. *Storage requirement*. This is the memory space required for the table lookup data structure.
3. *Update time*. This is the time required to insert/delete a route prefix into/from the forwarding table.

Many IP table lookup algorithms have been proposed to solve LPM problems. For instance, Ruiz-Sanchez et al. classified a large variety of table lookup algorithms and compared their worst-case complexities of lookup latency, update time, and storage usage [17]. The binary trie is the basic data structure used in most IP lookup algorithms. The binary trie allows the search for LPM to be in a bit-by-bit fashion. It is mostly implemented using linked lists in which each trie node has left and right pointers pointing to its left and right subtries, respectively. We cannot store all the nodes of the binary trie in a sequential array and then apply the binary search because there is no mechanism to compare two prefixes of different lengths.

In this paper, we shall propose two new IP lookup algorithms called *binary prefix search* and *multiway prefix search* based on a new mechanism to sort the prefixes in the forwarding table. Our goal is to store the prefixes in a linear array, and thus a faster search speed and a smaller memory requirement can be achieved. By generating less than N *auxiliary prefixes* in a routing table of N prefixes, the naïve binary search can be applied. As a result, the worst-case lookup complexity of $\log(N)$ can be obtained, and the storage complexity of the proposed binary prefix search can be as good as $O(N)$. In order to further reduce the storage requirement, we also propose a 32-bit representation to encode prefixes of any length. The comparisons and matching operations can be performed efficiently by using the 32-bit CPU instructions. The proposed binary prefix search can be easily extended to a multiway search scheme. Using a special array index technique, our multiway prefix search can achieve the maximum degree of the tree which is up to 17 and 33 using 32-byte and 64-byte cache lines, respectively. Therefore, the proposed multiway scheme can achieve at most four and three memory accesses for an IP lookup based on a forwarding table containing more than 120K route prefixes. These simulation results show that the proposed prefix search algorithms perform better than other existing IP lookup algorithms.

The rest of the paper is organized as follows. Section 2 describes existing IP lookup schemes, and Section 3 illustrates the basic data structure and the detailed lookup algorithms proposed in this paper. Section 4 shows the efficient encoding schemes for prefixes, and Section 5 improves the lookup performance by extending the proposed algorithms onto the fast cache architecture. Section 6 presents the results of the performance comparisons using real routing tables, and finally, the last section gives the concluding remarks.

2. Existing schemes and discussions

In this section, we classify the existing schemes based on three aforementioned metrics, namely, search speed, update speed, and memory requirement, instead of their data structures. These three metrics are the most important metrics used for evaluating different IP lookup schemes. However, it should be noted that improving one may significantly degrade another. A comprehensive review on the existing IP lookup schemes before 2001 can be found in [17]. In this paper, we are interested in the software-based lookup schemes that can be implemented in routers using network processors. We will not address designs with special hardware supports such as Ternary content-addressable memory (TCAM) [24] and pipelined ASIC-based engines [1].

2.1. Schemes for optimizing search speed

Ideally, we can use 4 GB to store the 1-byte port numbers of all IP addresses. Thus, only one memory access is needed for a lookup. However, the prefix update speed is slow, and the memory requirement is too high.

Multibit tries [20,18] basically avoid consuming too much memory by using hierarchical data structures. A multibit trie in which each node has 2^k children is called a k-bit trie, and k is called the stride of the trie. A k-bit trie can speed up the search performance by inspecting not just one bit but k bits at a time in an IP lookup operation. The stride size can be varied in any trie nodes at different levels. Therefore, if all nodes at the same level have the same stride size, it is considered as a fixed stride; otherwise, it is a variable stride.

Gupta et al. [6] proposed a two-level multibit trie with fixed strides such as hierarchical 24-8 and 16-16 tries. Using the same expansion technique, hierarchical 16-8-8 or 8-8-8-8 tries can be constructed. Larger strides result in faster searches, but the memory requirement for these is high, and the update process is slow because many entries need to be modified.

In another study [12], authors converted the prefix search problem to a range problem since a prefix is a special case of range. The prefixes are encoded by their start and end addresses, which are also called their endpoints. These endpoints are sorted and stored in a sequential array. Additional information such as "$>$" and "$=$" ports are employed to make the binary search on the sequential array work. The primary idea of the binary range search is to precompute the port numbers when the target IP is equal to one of the endpoints or is located between two consecutive endpoints. Waldvogel et al. [25] proposed a scheme (denoted by BS-Length) that performs a binary search on hash tables organized by prefix length. Both schemes in [12,25] need precomputation to obtain fast lookup performance.

2.2. Schemes for optimizing memory requirement

The small forwarding table (SFT) scheme [5] used a compressed version of 16-8-8 trie to reduce memory consumption. Nilsson and Karlsson [16] proposed a scheme called Level-Compressed (LC) trie which recursively transforms binary tries with prefixes into multibit tries. In LC trie, fill factor is used to determine if a level can be compressed, and the indices of the pre-allocated array are used as the pointers for linking nodes of different levels. Huang et al. [7] proposed a compressed 16-x lookup (C-16-x) scheme based on the run-length encoding technique. In another study [3], a hashing technique is proposed to reduce the memory size for the 8-8-8-8 tries. In [21], authors proposed a compression technique to compact the endpoints of the prefixes, and they used shared pointers to reduce the memory requirement.

2.3. Schemes for optimizing update

Sahni et al. proposed many dynamic IP lookup schemes in order to achieve the update complexity of $O(\log N)$ for a routing table of N prefixes. The binary tree on the binary tree scheme (PBOB) [13], the priority search tree scheme (PST) [14], and the collection of red–black tree schemes (CRBT) [19] are also studied in this paper. Their algorithms are basically based on balanced trees such as a segment tree, interval tree, and priority search tree.

2.4. Discussions

The balanced tree-based algorithms and 1-bit trie-based algorithms, such as the binary trie, use pointers to implement the desired data structures. Searches have to follow pointers to traverse the tree and thus involve expensive memory references. In addition, the primary memory of the

trie or balanced tree is used for storing pointers and other auxiliary information for speeding up the search and update processes. In general, the balanced tree and trie-based schemes have a high memory requirement. They also have a very good update performance but only a fair search speed.

The compression-based schemes mostly use arrays to implement the desired data structures. For example, the LC trie [16] and the SFT scheme [5] use arrays to implement the desired data structure, and thus the update process cannot be done without reconstructing the whole data structure. In general, the schemes using array implementation result in a faster search speed, and the memory requirement is low. However, the update speed is very slow because a lot of precomputations are needed.

Among all the existing algorithms, only the binary range search [12] can store the lookup data structure in a sequential array without other auxiliary data structures. Thus, its memory consumption is fair, and the search speed is fast. Similar to compression-based schemes, the update speed is very slow because precomputations such as shifting array elements for allocating spaces for new entries are needed.

In this paper, we will propose a new encoding format to store prefixes, which is simpler than the method proposed by Yazdani and Min [23]. Based on the new encoding format, prefixes can be compared and sorted easily. Auxiliary prefixes are generated in order to allow the prefixes to be stored in the sequential array, and thus the naïve binary search can be applied to find the LPM. Since the proposed scheme uses array implementation that results in a slow update process, we improve the update speed by using a 16-bit segmentation table as used in other studies [4–7,16]. The 16-bit segmentation table allows the search to be performed on the prefixes belonging to only one segment. The searches for IP addresses belonging to other segments are not affected by the ongoing update process.

3. Proposed data structure

Below are the notations and terminology used in this study.

$b_{n-1}\ldots b_0/l/p$: the *length format* of prefixes. It represents a prefix of length l associated with a next port number p in the n-bit address space. In IPv4, notation $b_{n-1}\ldots b_0/l$ or d3.d2.d1.d0/l will be used when no confusion is incurred.

$b_{n-1}\ldots b_0/m_{n-1}\ldots m_0/p$: the *mask format* of prefixes. It is similar to the length format except that n-bit mask $m_{n-1}\ldots m_0$ is used. For the latter, there exists an index i for $n-1 \geqslant i \geqslant 0$ such that $m_j = 1$ for all $j \geqslant I$, and $m_j = 0$ otherwise. Notation $b_{n-1}\ldots b_0/m_{n-1}\ldots m_0$ or d3.d2.d1.d0/m3.m2.m1.m0 for IPv4 will be used when no confusion is incurred.

$b_{n-1}\ldots b_i{*}\ldots{*}/p$: the *ternary format* of prefixes. It represents a prefix of length $n-i$ which is associated with a next port number p and $b_j = 0$ or 1 for $n-1 \geqslant j \geqslant i$. When we use $t_{n-1}\ldots t_1 t_0$ as the ternary format of a prefix, where $t_i = 0$, 1, or $*$ (don't care), we must follow the rule that if t_k is $*$, then t_j must also be $*$ for all $j < k$. For simplicity, a single don't care bit is used to denote a series of don't care bits, e.g., prefix 1$*$ denotes 1$****$ in the 5-bit address space.

Prefix enclosure: A prefix is said to be enclosed or covered by another prefix if the address space covered by the former is a subset of that covered by the latter. We use B $\subseteq$ A or A $\supseteq$ B to represent that prefix B encloses prefix A, where $\subseteq$ or $\supseteq$ is the enclosure operator.

Disjoint prefixes: Two prefixes are said to be disjoint if none of them is enclosed by the other.

It is known that the binary search works only for the sorted lists. Therefore, we must have a mechanism to compare prefixes. Comparing prefixes is not easy because the lengths of the prefixes may vary. In this section, we will first propose a systematic method to compare prefixes of various lengths. Also notice that a straightforward application of the binary search on a sorted list of prefixes may not work because the binary search scheme may direct the search to the places that are far from the original LPM corresponding to the target IP. To solve the problem, we will insert additional prefixes called *auxiliary prefixes* at some points in the sorted list. These points are the eventual places toward which the binary search moves. These auxiliary prefixes inherit the same routing information of the original LPM. Although the search may go away from the original LPM, the match will be found on one of these auxiliary prefixes.

Now, we first introduce our definition of comparing two prefixes in order to sort a list of prefixes of various lengths based on the ternary format.

Definition 1 (*Prefix comparison*). The inequality $0 < * < 1$ is used to compare two prefixes in the ternary format.

Fig. 1 shows a list of 15 sorted prefixes in the 8-bit address space by the above prefix comparison rule. If the same table is illustrated in the form of a binary trie, we can see that the relative positions of these sorted prefixes actually agree with the projected positions of the prefixes on the horizontal line. Let us study an example to see why performing a binary search on the list of sorted prefixes may encounter a failure. Consider the binary search operation for address $Dst = 01011000$ in Fig. 1. The first prefix to be compared is the middle one which is B = 01∗. Although B matches *Dst*, we cannot ensure that B is the LPM. Prefix B is temporarily stored, and the search continues. Since *Dst* is smaller than B, the search will continue on the list of GEHLKOF. Now, the middle one L = 01001100 is compared with *Dst*. Since *Dst* is larger than L, the list of KOF is searched. The search ends after prefixes O and K are checked. Since prefix K does not match the address *Dst*, only prefix B appears as the final result. Obviously, this result is incorrect since the LPM should be F = 01011∗. Moreover, prefix F did not get any chance to be examined in the process of the binary search.

By carefully analyzing the binary search on the sorted prefixes, we found that the main cause of the failure comes from the enclosure property between prefixes of different lengths [23,9]. In the above example, prefix O is enclosed by F. The search failure for address 01011000 is caused by the absence of a prefix that covers 01011000 and is also smaller than prefix O.

We did not try to revise the binary search operation in such a way that the search will end up locating F as the final match. What we did was to generate some auxiliary prefixes that inherit the routing information of the original LPM (e.g., F) and put them where the binary search operations can find them.

For example, if we insert an auxiliary prefix 01011000 inheriting F's routing information, then

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Port	G	E	H	L	K	O	F	B	A	N	I	M	J	D	C
Prefix	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
	0	0	0	1	1	1	1	1	0	1	0	1	0	1	1
	0	0	1	0	0	0	0	∗	∗	1	1	1	1	0	0
	1	1	1	0	0	1	1			1	1	1	1	1	∗
	0	∗	0	1	1	1	1			0	0	0	0	∗	
	∗		0	1	1	0	∗			0	0	0	1		
			∗	0	0	0				0	1	1	0		
				0	∗	1				1	∗	1	∗		

Fig. 1. List of sorted prefixes based on the rule of $0 < * < 1$.

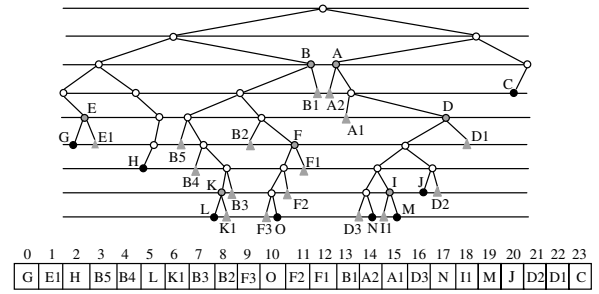


Fig. 2. The full tree expanded from the binary trie for prefixes in Fig. 1.

the search operation for address 01011000 will be successful. Therefore, it is feasible to split prefix F into two parts such that both sides of prefix O are covered. A simple solution is to remove all the enclosures by making the binary trie a full tree.[1]

The full tree expansion and prefix merges. The full tree expansion splits the enclosure prefixes into many longer ones and disjoints all the resulting prefixes. The sequential list of the sorted prefixes is then obtained by an inorder traversal of the full tree. Many auxiliary prefixes may inherit the same routing information of a common enclosure prefix. Some of these auxiliary prefixes from the same enclosure prefix may also be consecutively located. Therefore, these prefixes can be merged into one without affecting the correctness of the binary search because they inherit the same routing information. The merge operation is defined as follows.

Definition 2 (*Prefix merge*). The prefix obtained by merging a set of consecutive prefixes is the longest common ancestor of these consecutive prefixes in the binary trie.

Fig. 2 shows the full tree constructed by expanding the binary trie. The gray and black circle nodes represent the original prefixes in Fig. 1. The triangle nodes are the auxiliary prefixes expanded by the gray nodes. The full tree after the merge operations is shown in Fig. 3. Auxiliary prefixes B4 and B5 are merged into B7; B2 and B3 are merged into B6; F1 and F2 are merged back into F; A1 and A2 are merged back into A, and D1 and D2 are merged back into D. Keeping prefix E1 is equivalent to

[1] We follow the traditional notations used in most data structure and algorithms books in which a node either has no child or has two children in the full binary tree instead of the complete binary tree.

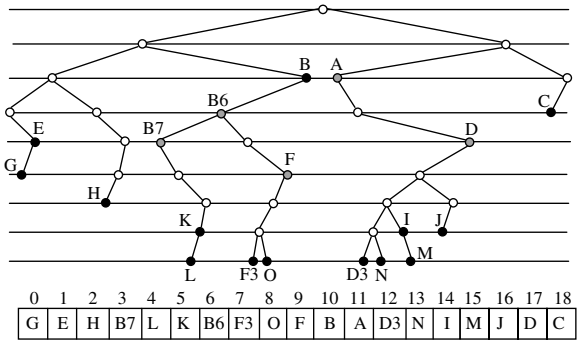


Fig. 3. The tree after merge operations.

keeping prefix E. We suggest keeping the original prefixes. Therefore, prefixes E, K, B, and I are retained.

The merge operations may reintroduce the enclosure property in the sorted list. Since only the auxiliary prefixes generated from the same original prefix may be merged, this time, the enclosure property does not cause any search failure. This is because the enclosure property that the final two prefixes examined in the last stage of the binary search may be the LPM. Therefore, when the target IP is located between two prefixes, the LPM will be the longer prefix if both prefixes match the target IP.

The following two lemmas illustrate how the merge operations can reduce the number of auxiliary prefixes and the relative position of the longest common ancestor prefix of two disjoint prefixes.

Lemma 1. *For a prefix that is enclosed by its immediate parent prefix, one auxiliary prefix, at most, which is inherited from the parent prefix may be generated based on the full tree expansion with merging.*

Proof. Assume a prefix y covers k consecutive original prefixes x_1 to x_k. If there is a region on the left side of x_1, a region between x_i and x_{i+1} (for $i = 1$ to $k-1$), or a region on the right side of x_k that is covered by y, then the full tree expansion must generate many auxiliary prefixes from y. These auxiliary prefixes are consecutive and thus can then be merged into their longest common ancestor. At most, there exists $k+1$ such regions, and thus at most, $k+1$ auxiliary prefixes may be generated from y. However, one of the auxiliary prefixes is the same as y. As a result, k auxiliary prefixes from y are newly generated at most. We can conclude that one prefix covered by its parent prefix contributes at most one auxiliary prefix based on the full tree expansion with merging. Thus, the lemma follows. □

The following lemma shows the relative positions between two disjoint prefixes and their longest common ancestor.

Lemma 2. *The longest common ancestor of two disjoint prefixes must be located between them in the sorted list.*

Proof. Let two disjoint prefixes be $A = b_{n-1}\ldots b_k a_{k-1}\ldots a_i{*}\ldots{*}$, and $C = b_{n-1}\ldots b_k c_{k-1}\ldots c_j{*}\ldots{*}$. Since they are disjoint, we assume $a_{k-1} = 0$, and $c_{k-1} = 1$. The longest common ancestor of these two prefixes must be $B = b_{n-1}\ldots b_k{*}\ldots{*}$. Based on Definition 1, we have $A < B < C$. Thus, the lemma follows. □

The merge operation involving many prefixes may be time consuming. The following lemma shows an efficient operation in which merging the leftmost and rightmost prefixes is sufficient to find the longest common ancestor of a list of consecutive prefixes.

Lemma 3. *The longest common ancestor of a list of consecutive prefixes is the same as that of the leftmost and the rightmost prefixes in the list.*

Proof. Consider a list of three consecutive prefixes l, m, and r that represents the prefixes on the left, in the middle, and on the right of the list. We first show by contradiction that the longest common ancestor of prefixes l and m is not disjoint from that of prefixes l and r. Let prefixes lm and lr be the longest common ancestors of prefixes l and m and prefixes l and r, respectively. Assume prefixes lm and lr are disjoint. Thus, any prefix enclosed by lm must also be disjoint from a prefix enclosed by prefix lr. This implies that the prefix l is disjoint from itself. Therefore, the assumption is contradictory. We can conclude that one of the prefixes lm and lr is enclosed by the other. If prefix lm is the same as prefix lr, then we are done. Therefore, we can only consider the case where lm ≠ lr.

Next, we show that prefix lm is enclosed by lr by using again the contradictory proof. Assume that prefix lr is enclosed by prefix lm. Let p be any prefix p enclosed by lr. From Definition 1, we know that p < lm or lm < p. Consider the case where lm < p. We can let p = l because l is one of the prefixes enclosed by lr. Therefore, we have m < lm < l. In short, m < l contradicts the original assumption.

Now, assume $p < lm$. We can let $p = r$ and get $r < lm$ because r is one of the prefixes enclosed by lr. Since $l < lr < r$, we have the condition of $l < lr < r < lm < m$ that in turn contradicts the condition of $r < m$. Therefore, the assumption that prefix lr is enclosed by prefix lm is also contradictory.

We now need to prove that the longest common ancestor of prefixes m and r must be enclosed by that of prefixes l and r. The proof is similar to that of the previous case. Thus, the lemma is proved. □

Obviously, the complexity of a merge operation is $O(n)$ in the n-bit address space. The overall merge operation can be done by scanning the prefixes one by one from the smallest prefix in order to group consecutive prefixes containing the same route information. Thus, the overall complexity is $O(n \times N)$. Now, we prove that the number of auxiliary prefixes generated by the proposed method is less than N for the routing table of N original prefixes.

Theorem 1. *The total number of prefixes in the proposed sorted prefixes is less than* $2N$, *where* N *is the original number of prefixes in the routing table.*

Proof. In the set of N prefixes, there are at most $N - 1$ prefixes that can be covered by a parent prefix. Therefore, based on Lemma 1, $N - 1$ auxiliary prefixes can be generated at most. In this case, the theorem is proved. □

Building and updating the sequential list. Now, we formally implement the building process of the proposed binary prefix search. Instead of performing the inorder traversal in the binary trie and then merging, we speed up the building process by performing the merging process and the inorder traversal in the binary trie at the same time. Fig. 4 shows the pseudo-code of the implementation. Lines 1–8 and 25–30 demonstrate the merging operations before and after processing a subtrie, respec-

```
// List stores the resulting sequential list of prefixes by the full tree expansion with merges
// EncStack denotes enclosure stack which stores the prefixes that covers the prefix
// currently processed and APQueue denotes auxiliary prefix queue
inorder_traversal_build_list(TrieNode node)
Begin
01  If node is Original_Prefix Then
02      If APQueue is not empty Then
03          Merge the auxiliary prefixes in APQueue into prefix MP
04          If the parent of MP is not equal to the previously added prefix Then
05              Add MP in List and reset APQueue
06      If the node is a leaf trie node Then
07          Add the node's prefix in List and reset APQueue
08      Push node's prefix in EncStack
09  If the node's left pointer is not null Then
10      Recursively call inorder_traversal_build_list using node's left pointer
11  Else { /* generate an auxiliary prefix */
12      If EncStack is not empty and node is not Original_Prefix Then
13          Create the node's left child as the auxiliary prefix and Store it in APQueue
14  If node is Original_Prefix Then
15      If APQueue is not empty Then
16          Merge the auxiliary prefixes in APQueue into prefix MP
17          If the parent of MP is not equal to the node's prefix Then
18              Add MP in List and reset APQueue
19      Add the node's prefix in List
20  If node's right pointer is not null Then
21      Recursively call inorder_traversal_build_list using node's right pointer
22  Else { /* generate an auxiliary prefix */
23      If EncStack is not empty and node is not Original_Prefix Then
24          Create the node's left child as the auxiliary prefix and Store it in APQueue
25  If node is Original_Prefix Then
26      Pop an entry from EncStack
27      If APQueue is not empty Then
28          Merge the auxiliary prefixes in APQueue into prefix MP
29          Add MP in List and reset APQueue
End
```

Fig. 4. Building the sorted list for the proposed binary prefix search.

tively. Recursive calls into the left and right subtries, and the creation of auxiliary prefixes are performed in lines 9–13 and 20–24. It is possible that the auxiliary prefix after merging inherits the same routing information as its left or right neighboring prefix in the list. In this case, this merged prefix is not inserted into the sorted list as shown in lines 2–5 and 15–18.

When inserting or deleting a prefix, one might hope that it is possible to develop an update algorithm of worst case complexity $O(\log N)$ as the balanced tree-based schemes proposed in [13,14,19]. However, such update algorithms is not possible without the pointer implementation, which violates our intention of using array implementation in order to obtain a very fast search process. Therefore, there does not appear to be any update technique that uses array implementation that is faster than just building the sorted prefixes from scratch. In order to reduce the impact of the update process on the overall router performance, we propose to use a 16-bit segmentation table.

Searching the sequential list. Since each prefix in the sorted list is not a single address but a range of addresses, the binary search applied on the sorted prefixes must be modified. The formal implementation of the proposed binary prefix search (BPS) is shown in Fig. 5. The lookup process starts by calling Proposed_BPS(List, 0, max − 1, target_ip). For example, in Fig. 3, when the target IP is 01011000, the final two prefixes are B6 and F3 after four successive probes on prefixes F, L, F3, and B6. After matching B6 and F3, prefix F3 is determined to be the final LPM because F3 is longer than B6.

4. The prefix representation

We have proposed a novel scheme to sort the prefixes on which a binary search can be applied to find the LPM. Recall that the two most important operations in IP lookups are comparing two prefixes of various lengths and matching a prefix with a 32-bit IP address. We have previously defined the comparison of two prefixes in Definition 1. What we mean by *match* is to determine if a target IP belongs to the subnetwork represented by the prefix.

The comparison and match operations involve ternary operations because 0, 1, and * (the don't care bit) are checked. Using the binary ALU instructions provided by current processors, there is no efficient way to perform the ternary operations. In this section, we first propose a 33-bit binary representation to encode the prefixes of any length in the context of IPv4. The proposed representation can be straightforwardly extended to IPv6. Based on the 33-bit representation, the operations of comparing two prefixes and matching a prefix with the 32-bit target IP can be performed easily using the 32-bit wide instructions of the current processors. However, the efficiency of the comparison and matching operations is brought down by the fact that a prefix encoded by the 33-bit representation still needs two 32-bit storages. Therefore, we propose an optimized 32-bit representation to remedy this drawback.

There are two commonly used binary representations for the prefixes of different lengths in IPv4, namely, the mask format and the length format. Compared to the mask format, the length format

```
// List is an array of sorted prefixes after full tree expansion and merge.
// L and R are the left and right indices of List array, and IP is the target IP address.
// ⊇ is the enclosure operator
Proposed_BPS(Prefix List[], Index L, Index R, Address IP)
Begin
01    If (L+1 = R) Then
02       If (length(List[L]) ≥ length(List[R])) Then
03             If (List[L].addr ⊇ IP) Then Output port corresponding to prefix List[L];
04             If (List[R].addr ⊇ IP) Then Output port corresponding to prefix List[R];
05       Else
06             If (List[R].addr ⊇ IP) Then Output port corresponding to prefix List[R];
07             If (List[L].addr ⊇ IP) Then Output port corresponding to prefix List[L];
08       Output defulat port;
09    M = floor( (L+R)/2);
10    If (List[M].addr = IP) Then Output port corresponding to prefix List[M];
11    Else If (List[M] < IP) Then recursively call Proposed_BPS(List, M, R, IP);
12    Else recursively call Proposed_BPS(List, L, M, IP);
End
```

Fig. 5. Algorithm for the proposed binary prefix search on a list with enclosure property.

has the advantage in that only 5 bits are required for the length in IPv4 instead of a 32-bit netmask. Matching can be done easily by the following steps. The IP part of the prefix is first XORed with the target IP address. Then the result is ANDed with the netmask (or right shifted 32 – length bits) if the mask (or length) format is used. If the final result is zero, then the match is found.

However, no matter which format is used, comparing two prefixes is not easy. Consider two prefixes m1.m2.m3.m4/mask1 and n1.n2.n3.n4/mask2 with lengths len1 and len2, respectively. Assuming mask1 $\leqslant$ mask2, we perform two AND operations, m1.m2.m3.m4&mask1 and n1.n2.n3.n4&mask1. If these two ANDed results can allow us to determine which of these two prefixes is larger, then we are done. The difficulty arises in the situation in which m1.m2.m3.m4&mask1 equals n1.n2.n3.n4&mask1. We must perform an additional check to see if the (31-len1)th bit of n1.n2.n3.n4 is 0 or 1. If the (31-len1)th bit of n1.n2.n3.n4 is 0, then m1.m2.m3.m4/mask1 is larger than n1.n2.n3.n4/mask2. Otherwise, m1.m2.m3.m4/mask1 is smaller than n1.n2.n3.n4/mask2. All these operations are complex to implement. We will introduce our proposed representation and demonstrate the efficient implementation of prefix comparisons and matching using 32-bit CPU instructions as follows.

The 33-bit prefix representation: A simple 4-bit case is studied first. There are 31 different prefixes of various lengths in a 4-bit address space. Using binary representation to identify 31 different items, we need a 5-bit address space. There are many ways to map these 31 prefixes to the 5-bit address space. We select a mapping that is consistent with Definition 1. This mapping preserves the relative positions of the prefixes in the binary trie projected on the horizontal line. The address 00000 is unused. The binary representation for prefixes of various lengths can be easily extended to the IPv4's 32-bit or IPv6's 128-bit address space. We formally define the binary representation of a prefix in an n-bit address space as follows.

Definition 3 (*Definition of* $(n+1)$*-bit prefix representation in the n-bit address space*). For a prefix of length i, $b_{n-1}\ldots b_{n-i}*$, where $b_j = 0$ or 1 for $n-1 \geqslant j \geqslant n-i$, its binary representation is $b_{n-1}\ldots b_{n-i}10\ldots0$ with $n-i$ trailing zeros.

Consider two prefixes 01* and 01010* in an 8-bit address space. From Definition 1, we have 01* > 01010*. Based on Definition 3, we represent 01* and 01010* as 011000000 and 010101000, respectively. Thus, it is easy to use a 9-bit binary comparison operation to determine that 01* is larger than 01010*. However, for the 32-bit address space, a 33-bit binary comparison operation is required. It means two 32-bit binary comparison operations may be needed in the worst case since only 32-bit arithmetic and logic operations are available in current 32-bit processors. It also means that two 32-bit memory reads are needed if the size of registers is 32 bits. In addition, two 32-bit words are needed to store a prefix using 33-bit representation. To solve the above problems, an optimized 32-bit representation is proposed.

The 32-bit prefix representation. Based on the 33-bit prefix representation, 33 bit-by-bit comparisons or two 32-bit comparisons can fully determine the order of the prefixes. If there is no prefix of length 32, we can use only 32 bits by ignoring the least significant bit that must always be zero. Notice that the smallest subnet in classless inter-domain routing (CIDR) is at least 2 bits long that contains one network address, one broadcast address, and two usable IP addresses. Thus, assuming there is no prefix of lengths, 31 and 32 are reasonable. Notice that the routing cache design using CPU caches [4] make the same assumption. However, by investigating the routing tables of current routers available on the Internet, there is a small number of prefixes whose lengths are 31 or 32. The prefixes of length 32 may come from the direct connections between the router and the computers or devices. The prefixes of length 31 may be configured for testing purposes or result from configuration errors in one router, and are inadvertently passed into the inter-domain routers [8]. Therefore, unless the prefixes of lengths 31 and 32 are filtered out, distinguishing two prefixes by using only 32 bits is the main problem to be solved.

The ambiguity caused by removing the least significant bit from the 33-bit representation will be solved by a special layout design in the sorted list of prefixes. For simplicity, an 8-bit address space is used to illustrate our idea. Consider two prefixes P1 = 01001100/port1 and P2 = 01001*/port2. The 9-bit prefix representations of P1 and P2 are 010011001 and 010011000, respectively. The first 8 bits cannot distinguish P1 from P2. In general, when two prefixes have the same first 8 bits, one of them must be of length 8, and the other may be of any length except 8. We call one of these two prefixes as the *buddy prefix* of the other. Any prefix may

have its buddy prefix coexisting in the sorted list. In order to make the binary search on the sorted list of 8-bit prefixes correct, every time a prefix is matched against the target IP, we need to do a further check if its buddy prefix also exists on its left or right side. This additional check significantly slows down the search process.

We solve this problem by means of the following rule: only the prefix of length $n-1$ is allowed to have a buddy prefix of length n coexisting in n-bit address space. This rule is achieved by the following conversion technique. Based on this technique, we only need to check if a buddy prefix coexists when the prefix of length n or $n-1$ is examined. This conversion technique may generate at most the same number of additional prefixes as that of the original prefixes of length n.

Definition 4 (*Definition of prefix conversion in the n-bit address space*). (a) If a prefix of length n, $b_{n-1}\ldots b_1 0/n/p_1$ exists, it is converted to $b_{n-1}\ldots b_1 0/n-1/p_1$, and prefix $b_{n-1}\ldots b_1 1/n/p_2$ created if it does not exist in the list, where p_2 is the port of the prefix that covers $b_{n-1}b_{n-2}\ldots b_1 1$. (b) If prefix $b_{n-1}\ldots b_1 1/n/p_1$ exists in the list, then $b_{n-1}\ldots b_1 b_0/n-1/p_2$ is created if it does not exist in the list, where p_2 is the port of the prefix that covers $b_{n-1}b_{n-2}\ldots b_1 0$. (c) If only $b_{n-1}\ldots b_1 0/n-1/p_1$ exists in the list, do nothing.

Fig. 6 shows how the prefix conversion operations work. Based on the above definition, prefix $b_{n-1}\ldots b_1 0/n/p_1$ is not allowed to occur in the list since its buddy prefix with a length shorter than $n-1$ may also exist. Fig. 6(a) shows that when $A = b_{n-1}\ldots b_1 0/n/p_1$ exists, it is first converted to $b_{n-1}\ldots b_1 0/n-1/p_1$. If prefix $b_{n-1}\ldots b_1 0/n-1/p_2$ already exists, it will be converted to $B = b_{n-1}\ldots b_1 1/n/p_2$. Otherwise, prefix $B = b_{n-1}\ldots b_1 1/n/p_2$ will be created, where p_2 is the port number of the longest prefix that covers B. Fig. 6(b) shows that when $A = b_{n-1}\ldots b_1 1/n/p_1$ exists, prefix $B = b_{n-1}\ldots b_1 0/n-1/p_2$ is created when B does not exist, and p_2 is the port number of the longest prefix that covers B. Fig. 6(c) shows that if only $A = b_{n-1}\ldots b_1 0/n-1/p_1$ exists, no conversion is needed.

If both $b_{n-1}\ldots b_1 0/n-1/p_1$ and $b_{n-1}\ldots b_1 1/n/p_2$ exist after conversion, the latter is stored as $0\ldots 0/p_2$ if the 32-bit format is used. When $0\ldots 0/p_2$ is found in the search, we know that a prefix and its buddy prefix must coexist in the list. However, when the search encounters $b_{n-1}\ldots b_1 1/p_1$ which is of length $n-1$, an additional operation must be performed to check if its buddy prefix (stored as $0\ldots 0/p_2$) coexist in the list. If only $b_{n-1}\ldots b_1 0/n-1/p_1$ exists, it must be stored as $b_{n-1}\ldots b_1/1/p_1$ instead of $0\ldots 0/p_1$. Fig. 7 illustrates the full tree after conversions from Fig. 3. For example, prefixes L and K are converted to L0 and K1. Thus, when we perform a comparison with K1 which is stored as 0...0, we only know K1's length that is 8 but not its value. The value of K1 is known only after examining L0.

Now we consider how to match a prefix encoded by a 32-bit representation against an IP using the 32-bit instructions of the current processors. The matching process can be done by the following steps. First, the position of the least significant set bit in the 32-bit prefix must be computed to determine the length of the prefix. If the least significant

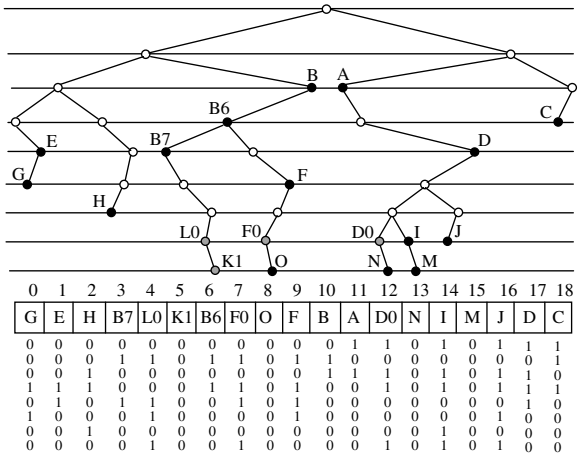


Fig. 7. The list of sorted prefixes after the prefix conversions.

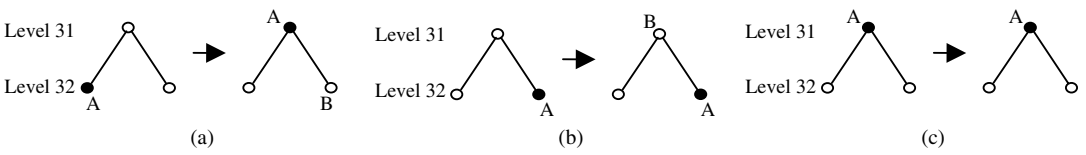


Fig. 6. Illustrations of different cases in prefix conversion rule.

set bit is on bit 0, then the length of the prefix is $n-1$, and thus we need to check if its buddy prefix of length n also exists. The computation time for this operation seems dependent on the position of the least significant set bit. Fortunately, there exists an instruction called bit scan forward (BSF) in the Intel processor family [10] starting from Intel 80386 which can perform the required task. Second, let i be the position of the bit computed by BSF on the 32-bit prefix. Thus, we compute (P XOR IP) $\gg (i+1)$, where "$\gg$" denotes the right shift operation. If it is zero, then prefix P matches IP.

The lookup process using the 32-bit representation is similar to the algorithm shown in Fig. 5. The only extra step required is that when the prefix P with address zero or the prefix of length $n-1$ is examined, the existence of its buddy prefix must be checked. Since the number of prefixes with length 32 is small and is equal to the number of prefixes with address zero in the list, the impact on the lookup performance will be negligible.

5. *m*-Way search tree using cache lines

Modern general-purpose CPUs support various kinds of caches for speeding up the data processing. Most modern CPUs have the cache line size of 32, 64, or 128 bytes. The primary idea of using cache lines for searching is to reduce the number of searches in the memory by performing the searches in cache lines. Therefore, putting as much search information in a cache line is the key factor to design efficient m-way search tree algorithms.

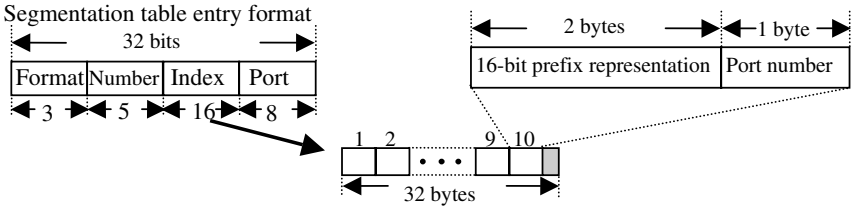
We use a 16-bit segmentation table because the remaining 16 bits from the prefixes are byte aligned and can be efficiently fit to cache lines of 32 or 64 bytes. Also, the update process only causes one segment to be reconstructed. The basic element in the sequential list takes 3 bytes to store the remaining 16-bit of a prefix and its associated 8-bit port number, which is denoted as a 2-tuple (prefix, port). We develop two basic encoding schemes to encode the prefixes or their corresponding port numbers in a memory block. The first scheme encodes 10 3-byte 2-tuples in a 32-byte cache block where 2 bytes are left unused. The second scheme encodes 16 16-bit prefixes in a 32-byte cache block. Based on these two encoding schemes, different hierarchical structures are proposed as the m-way prefix search trees.

The hierarchical structure of prefixes is designed in a block-by-block fashion, where a block size is the same as that of a cache line. The base of the block array is assumed to be known globally. Thus, instead of the global address, the indices of the memory blocks are used to access the prefixes stored in the list. We use the binary search inside the cache line. The entry in the 16-bit segmentation table is of size 32 bits, and it contains a Format field, a Number field recording the number of prefixes in the segment, an Index field or prefix field, and a Port field. The data structures of the different entry formats in the 16-bit segmentation table will be described in detail when necessary.

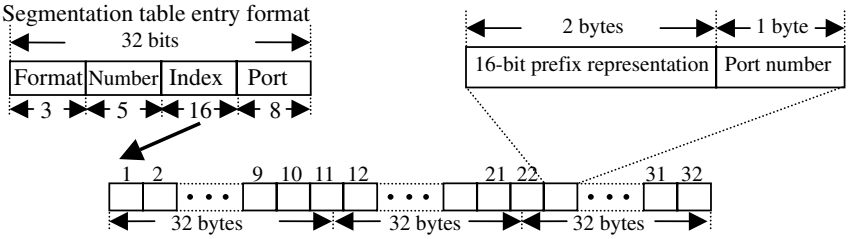
We first consider two special cases when there is zero or one prefix in the segment. If there is no prefix of length longer than 16 in the segment, the lookup operation should return the port number belonging to that segment or the global default port number. All the fields in the corresponding entry of the segmentation table are set to zero except for the Port field. If there is only one prefix of length longer than 16 in the segment, the proposed 16-bit representation for the prefix and the corresponding port number are stored in the entry of the 16-bit segmentation table. These two cases are categorized as format 0 segment.

If there are k prefixes for $2 \leqslant k \leqslant 10$ in a segment, then these k prefixes can be stored in a memory block of size 32 bytes. The data structures of various segments are illustrated in Fig. 8. The *Format* field (3 bits) stores the format number which is 1. The *Number* field (5 bits) stores the value k. The *Index* field (16 bits) stores the index of the memory block for this segment. The *Port* field (8 bits) indicates the local default port in this segment. Storing default port number avoids generating too many auxiliary prefixes. We can see that two memory accesses are required to obtain the output port number: one for accessing the segmentation table and another for accessing the memory block.

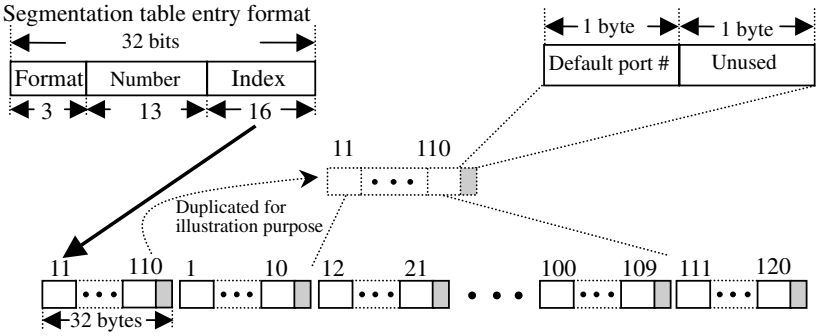
If there are more than 10 prefixes in the segment, then more than two memory accesses are required. For a segment containing 11–32 prefixes as shown in Fig. 8(b), exactly 3 blocks are needed. The binary search inside these three blocks takes at most two memory accesses. Overall, the worst case search time is three memory accesses. The data structure of the format 2 entry is the same as that of the format 1. The binary search on the prefixes stored in memory blocks can be applied directly. If there are k prefixes for $33 \leqslant k \leqslant 120$ in a segment (format 3), a 2-level 11-way search tree is constructed as shown in Fig. 8(c). This 2-level 11-way search tree is physically laid out as a sequential list. The first



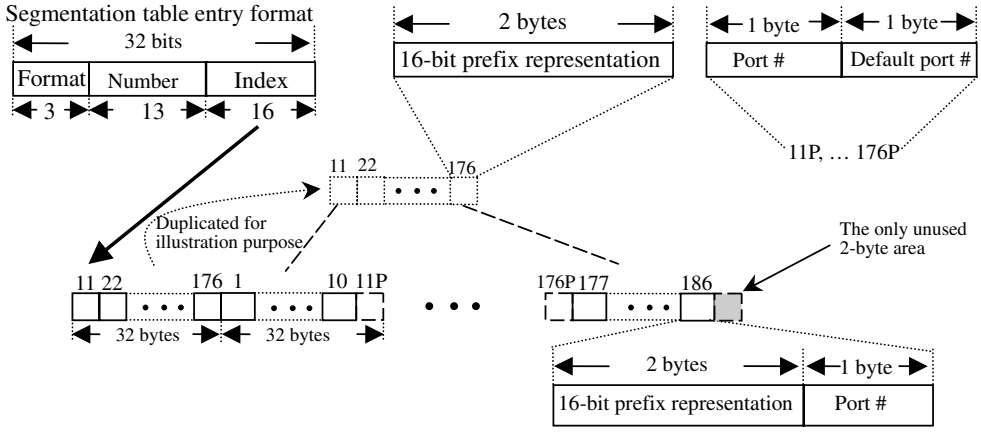
(a) format 1: Structure of the segment containing 2 to 10 prefixes.



(b) format 2: Structure of the segment containing 11 to 32 prefixes.



(c) format 3 : A 2-level 11-way prefix segment containing 33 to 120 prefixes.



(d) format 4: A 2-level 17-way prefix segment containing 121 to 186 prefixes.

Fig. 8. Structures of the segments.

block in the sequential list is the root block that is duplicated in the figure for illustration purpose. The root block stores the prefixes whose index is multiples of 11. Since the 32-bit entry of the segmentation table is limited, the default port number is moved to the unused 2-byte area.

To pact more prefixes in a block, another hierarchical structure (format 4) based on the second

encoding scheme is proposed as shown in Fig. 8(d). At most, 186 keys can be stored in a two-level 17-way search tree. The root block encodes 16 keys (16-bit representation of prefixes) without the associated port numbers. The port numbers corresponding to the prefixes in the root block are stored in the unused 2-byte area in the second level. In addition, we also distribute the default port number over the unused area of every second level block. In Fig. 8(d), the entries denoted by indices 11P,..., 176P are the original unused 2-byte areas that now store the port numbers of the 11th,..., 176th prefix in the list, respectively, and the default segment port number. Since the default port is not stored in the root block, searches in a 2-level 17-way search tree always take three memory accesses.

The hierarchical structures based on the proposed encoding schemes can be extended to more than 2 levels. As we know, the cache line size can be 32, 64, or 128 bytes, and the results are shown in Table 1. We briefly describe formats 5, 6, and 7 by using the block of size 32 bytes as follows. In format 5, the blocks in all the levels contain 10 prefixes and their associated port numbers. In format 6, only the root block which records 16 prefixes with the associated port numbers are stored in the second-level blocks. In format 7, the root block and the second-level blocks which contain 16 prefixes without associated port numbers are distributed over the third-level blocks accordingly.

Default port of a segment. We have shown that the delete and insert processes have the worst-case time complexity of $O(N)$ for a list of N prefixes. Using the 16-bit segmentation table allows us to reduce the worst-case update complexity to $O(N_{seg})$, where N_{seg}, much less than N, is the maximum number of prefixes among all 65,536 segments. The prefixes of length less than or equal to 16 are not used to generate auxiliary prefixes. The total number of the auxiliary prefixes is reduced. Thus, the search and update performance is improved.

Consider an example in which there are 11 original prefixes in one segment. Assume that one prefix encloses the other 10 disjoint prefixes. If we use the method proposed in Section 3 to build the sequential list, 10 auxiliary prefixes at most may be generated, and the total number of prefixes will be 21. Thus, the worst-case number of memory accesses will be three (one to the segmentation table and two to the list), assuming format 2. If we make the port of the enclosure prefix the default port, no auxiliary prefix is generated. Consider the 32-byte block and format 1. The port and the corresponding enclosure prefix can be stored in the Port field of the segmentation table entry and the unused 2-byte area of the root block. The worst-case number of memory accesses will be two (one to the segmentation table and one to the list). This technique can be applied to other formats.

For 32-byte cache lines, the maximum degree of the tree in the proposed multiway schemes is 11 or 17 depending on whether or not the port numbers are stored along with the prefixes in the memory blocks. The maximum degree of the tree becomes 22 or 33 and 43 or 65 for the cache lines of 64-byte and 128-byte, respectively. This is a big improvement over the multiway range search and multiway range tree. Notice that the maximum degree of the multiway range tree [22] is smaller than that of the multiway range search because each cache line needs to record the heads of the equal list and the span list. Please refer to the definitions of equal and span lists in [22]. Therefore, the multiway range tree is deeper than the tree based on the multiway

Table 1
The numbers of prefixes in different segment formats

Format	Tree level	n = Number of entries in the segment			Block index/16-bit prefix	Port
		CLS	32	64		
0	0	0	0	0	0	Default port #
0	0	1	1	1	Prefix	Port #
1	1	$2 - k_1$	2–10	2–21	Index	Default port #
2	2	$k_1 + 1$–CLS	11–32	22 ~ 64	Index	Default port #
3	2	$CLS + 1 - (k_1 + 1)^2 - 1$	33–120	65–483	Index	n/a
4	2	$(k_1 + 1)^2 - (k_2 + 1)(k_1 + 1) - 1$	121–186	484–725	Index	n/a
5	3	$(k_2 + 1)(k_1 + 1) - (k_1 + 1)^3 - 1$	187–1330	726–10,647	Index	n/a
6	3	$(k_1 + 1)^3 - (k_2 + 1)(k_1 + 1)^2 - 1$	1331–2057	10,648–15,971	Index	n/a

CLS = cache line size in bytes, $k_1 = \lfloor CLS/3 \rfloor$, $k_2 = \lfloor CLS/2 \rfloor$.
n/a = Not applicable, $\lfloor x \rfloor$ = floor of value x.

range search. Although the child-sibling linear array is used in [22] to reduce the overhead of storing child pointers, the memory usage must also be increased.

6. Performance evaluation

In this section, we first use a real large routing table to analyze the impact of the auxiliary prefixes generated from the proposed scheme, and then compare the result with the binary range search [12]. Second, we conduct trace-driven simulations to evaluate the lookup times and the sizes of the memory required for the proposed schemes, and compare them with other existing lookup schemes. Our experiments are performed on a Linux Redhat platform with a 2.4-GHz Pentium IV processor containing 8 KB L1 and 256 KB L2 caches of 64-byte cache lines. Since it is almost impossible to obtain the actual IP traffic being routed through the router at the time when the routing table is logged, we use a simulated IP traffic, which is described as follows. Assume the routing entries in the routing table follow the length format. The simulated traffic is constructed by first removing the length and port of the routing entries, and then performing a random permutation. The simulated traffic assumes that every prefix in the routing table has the same probability of being accessed. The same method is also used in [16,17]. The detailed simulation designs can be referred to [3].

We have proved in Theorem 1 that the number of prefixes in the sorted list after merging operations is less than $2N$, where N is the number of original prefixes in the routing table. Therefore, the worst-case complexities of the proposed binary search scheme in the lookup time, update time, and memory usage are similar to those of the range search schemes [12]. However, there are subtle differences in terms of memory requirements and lookup times between the proposed prefix searches and the range searches which are described as follows. Consider a routing table of N prefixes. The binary range search in fact needs a sequential array of almost $2N$ elements, each of which uses 6 bytes to store the 32-bit endpoint address and two 1-byte port numbers (“>” and “=” ports). For Oix-120k table containing $N = 120{,}629$ original prefixes, the binary range search requires 232,887 endpoint addresses, as shown in Table 2. Without the 16-bit segmentation table, the full tree scheme and that with the merging operations need around $1.65N$ and $1.2N$ prefixes, respectively. Notice that each prefix only needs 5 bytes to store the 32-bit prefix and the 1-byte port number. Thus, from Table 2, the total amount of memory required for the binary range search, the proposed full tree scheme, and the proposed scheme with merging operations can be computed as 1365 KB, 975 KB, and 712 KB, respectively. The number of auxiliary prefixes decreases from 89,225 to 35,211 when the merge operations are performed. It accounts for 270 KB memory reduction.

The last row of Table 2 shows the total numbers of prefixes when a 16-bit segmentation table is employed. The total numbers of prefixes for the full tree scheme and that for the merging are 133,160 and 117,968, respectively. The latter is even smaller than that of the original prefixes. The main reason is that the original prefixes of length less than or equal to 16 are embedded into the segmentation table. However, the 16-bit segmentation table requires an additional memory of 256 KB.

Table 3 shows the worst-case numbers of memory accesses, the search time, the update time, and the amount of memory required for various schemes using the Oix-120k table. Unless otherwise stated, a 16-bit segmentation denoted by the suffix “−16” is assumed. The proposed prefix searches are generally better than most of the existing schemes. The memory reduction is mostly attributed to the 32-bit prefix representation. Specifically, the proposed prefix searches outperform the range searches both in memory consumption and the number of memory accesses for a lookup operation. The multiway prefix search has the same worst-case number of memory accesses as C-16-x scheme [7]. However, the C-16-x scheme consumes more memory. The amounts of memory required for the proposed binary prefix searches with or without a 16-bit segmentation table

Table 2
The numbers of endpoints and prefixes needed in the range search and proposed prefix search for the Oix-120k routing table

Sixteen-bit segmentation table	Original	Range search	Prefix search		Prefix search with merge	
		Total endpoints	Total prefixes	Auxiliary prefixes	Total prefixes	Auxiliary prefixes
Without	120,629	232,887	199,751	89,225	145,737	35,211
With	120,629	217,146	133,160	28,293	117,968	13,101

Table 3
Performance for the routing table of 120,629 prefixes (Oix-120k), where the proposed multiway and binary prefix searches are denoted by MPS and BPS, and suffix "−16" denotes the usage of a16-bit segmentation table

Scheme	Worst case # of memory references	Search time (T_s) in μs	Update time (T_u) in μs	Memory (*Mem*) in KB
BTrie-16	17	0.47	0.84	2447
C-16-x [7]	3	0.17	9.89	1147
BS-Length [25]	5	0.35	210	2315
SFT [5]	12	0.21	1454	650
BRS-16 [12]	4	0.28	9.56	1104
MRS-16 [12]	4	0.23	29.3	4695
BPS-16	4	0.16	6.05	601
MPS-16	3	0.17	17.82	954
PBOB-16 [13]	10	0.34	0.97	3550
PST-16 [14]	10	0.44	1.58	7932
CRBT-16 [19]	10	0.65	4.20	19,535

are even smaller than that of the small forwarding table scheme [5]. Notice that if the original prefixes are stored in a linear array, the linear search can be applied. Obviously, the search and update speeds will be very slow. However, this arrangement will have the lowest memory requirement. If we assume that each original prefix is stored in the length format (45 bits for a prefix), then the size of the memory required for the Oix-120k table is 663 KB.

Since the proposed prefix search and the range search are similar and both use array implementation, we also obtained the performance results of other routing table traces in different sizes to have a complete understanding of their differences. The results are summarized in Table 4. We can see that the performance improvement of the proposed prefix searches over the range searches in lookup times increases as the size of the routing table increases. For all the simulated routing tables, the binary and multiway prefix searches consume 25–48% and 72–81% less memory than the binary and multiway range searches, respectively. For Funet-40k, the lookup times of the proposed prefix searches are only a little better than their range search counterparts (2–12% less). However, for larger tables, the lookup times of the proposed prefix searches are 26–42% less than their range search counterparts.

We also ran the performance simulations for various IPv6 routing tables, where two of them are real small tables obtained from the Internet [2], and the other three bigger ones are generated synthetically with similar length distributions to the first two real tables. The results are shown in Table 5. In terms of lookup times, the proposed binary prefix search performs better than the binary range search when the routing table contains 10,000 entries or more. However, the proposed binary prefix searches consume significantly less memory than the binary range searches.

The main drawback of storing the sorted prefixes in a sequential list is its update process which requires a reconstruction of the whole list when inserting or deleting a prefix. Therefore, a feasible remedy for this is to use the 16-bit segmentation table in which the update process is reduced to reconstructing a sublist belonging to the segment of a prefix being inserted or deleted. As shown in Table 3, we measure the update time of the binary prefix search and compare it with other existing schemes. The update times are calculated with the

Table 4
Average lookup times in μs and amount of memory in KB required for the range searches and the proposed prefix searches with a 16-bit segmentation table

	Funet-40k [16] (41,709 prefixes)	Oix-80k [15] (89,088 prefixes)	Oix-120k [15] (120,629 prefixes)	Oix-160k [15] (159,930 prefixes)
Binary range	0.126 (525.5 KB)	0.234 (846.9 KB)	0.267 (1104 KB)	0.304 (1500 KB)
Multiway range	0.148 (1719 KB)	0.219 (3543 KB)	0.237 (4696 KB)	0.262 (6133 KB)
Binary prefix	0.123 (395.5 KB)	0.145 (558.5 KB)	0.159 (601.6 KB)	0.188 (843.3 KB)

Table 5
Normalized average lookup times in μs and amount of memory in KB for the binary prefix searches (BPS) over the binary range searches (BRS) using IPv6 tables

	V6table (274)	Asmap (593)	Random5k (5000)	Random10k (10,000)	Random20k (20,000)
Time ratio (BPS/BRS)	1.12 (0.188/0.168)	1.01 (0.214/0.212)	1.00 (0.236/0.236)	0.95 (0.253/0.266)	0.83 (0.282/0.34)
Memory ratio (BPS/BRS)	0.62 (5.3/8.6)	0.72 (13.68/19.1)	0.49 (86.0/175.5)	0.50 (176.6/350.5)	0.53 (370.1/ 700.7)

assumption that the insertion and deletion have equal probability. We can see that the update speeds of Btrie-16, PBOB-16, and PST-16 are among the best. The update time of the collection of red–black trees (CRBT) is a little bit worse because many trees need to be modified when a prefix is being inserted or deleted. The average update time of the proposed binary prefix search (BPS-16) is better than that of the binary range search (BRS-16) and the C-16-x scheme. The multiway versions of the range and prefix searches do not perform well in terms of update times, especially for the worst-case update times (not shown in the table), which are 50 times slower than the PBOB-16. Since the operations of inserting and deleting a prefix are very simple, the update times for the binary trie with or without a 16-bit segmentation are much faster than those of other schemes.

Integrated performance analysis. Since different lookup schemes have their merits in terms of search speed, update speed, and memory consumption, we try to analyze the performance of various schemes by considering all these three factors together in the performance test. We first propose a simple model to analyze the maximum number of lookups (N_s) that a lookup scheme can sustain in 1 s when there are N_u update packets to be processed in the same 1 s period. N_u can be up to a few hundreds to a few thousands in the real routing environment [8,11]. Assume that the search and update operations take T_s and T_u microseconds, respectively. The following equation gives the result:

$$T_s \times N_s + T_u \times N_u = 1,000,000. \quad (1)$$

To simplify our analysis, we assume $N_u = \alpha \times N_s$. Thus, we have

$$N_s = 1,000,000/(T_s + \alpha \times T_u). \quad (2)$$

To include the memory consumption in the analysis, we treat the maximum number of sustained lookups and the memory size as the performance and the cost of a lookup scheme. Thus, we use the performance-cost ratio ($\text{ratio} = N_s/Mem$) as the metric to compare all the schemes.

Table 6 shows the maximum number of lookups that can be performed using different lookup schemes for the Oix-120k table. Other tables of different sizes show similar results which are not given in this paper. We assume that an ideal scheme can be developed in such a way that it can achieve the best search speed and the update speed compared to all the existing lookup algorithms. We also assume that the ideal scheme needs the same size of memory as the original routing table with the assumption of length format. Therefore, the ideal scheme has the performance of $T_s = 0.1$ μs and $T_u = 0.8$ μs, and needs 663 KB of memory as computed earlier. The parameter α is set to 0.05–0.001. For the ideal case, it amounts to the update rate of 10k to 370k updates per second.

As shown in Table 6, the proposed binary prefix search performs best in terms of the maximum number of sustained lookups and the performance-cost ratio. The only exception is when the update rate is one-twentieth of the lookup rate which is equivalent to more than 100 k updates per second. This high update rate will not be expected in the near future. In terms of N_s, the C-16-x scheme [7] performs only a little worse than the proposed binary

Table 6
Integrated performance comparisons in terms of the maximum number of sustained lookups and the performance-cost ratio for the Oix-120k table

	N_s				Ratio			
α	0.05	0.01	0.005	0.001	0.05	0.01	0.005	0.001
Ideal	7,142,857	9,259,259	9,615,385	9,920,635	10,173	13,966	14,503	14,963
Btrie-16	1,953,125	2,090,301	2,108,815	2,123,864	798	854	862	868
BRS-16	1,485,884	3,219,575	3,769,318	4,365,668	1,346	2916	3414	3954
BPS-16	2,285,714	4,819,277	5,594,406	6,420,546	3903	8019	9308	10,683
SFT	13,712	67,893	134,147	611,658	21	104	206	941
C-16-x	1,115,449	3,341,129	4,451,369	6,063,178	972	2913	3881	5286
BS-len	92,165	408,163	714,285	1,785,714	40	176	309	771
PBOB-16	2,574,003	2,859,594	2,899,812	2,932,809	725	805	817	826
PST-16	1,933,804	2,203,634	2,242,751	2,275,059	244	278	283	287
CRBT-16	1,160,964	1,442,544	1,487,646	1,525,810	59	74	76	78

prefix search when the update rate is not high. However, the C-16-x scheme performs much worse than the proposed binary prefix search when the update rate is high. If we consider the performance-cost ratio, the C-16-x scheme performs consistently worse than the proposed binary prefix scheme. The small forwarding table (SFT) [5] and the collection of red–black trees (CRBT) [19] are two extreme cases whose performance-cost ratios exhibit very low values. SFT has a fast search speed and a low memory requirement, but its update speed is low. In contrast, CRBT has a very fast update speed, but its memory requirement is high, and the search speed is fair.

Limitation of the proposed schemes. With reference to the segmentation table in Fig. 8, our design supports eight different multiway segments. Therefore, three bits are sufficient for eight formats. Thirteen bits are proposed to represent the number of prefixes in a segment. Therefore, 8192 prefixes at most can be supported in one segment. We also use 16 bits to record the starting block number of the segment. The total number of memory blocks supported is up to 64K. If we assume that the average number of prefixes in a block is five, the supported routing table can have up to 320K prefixes, which is sufficiently larger than the size of the backbone routers currently employed on the Internet.

It is worth noting that the entry structure of the 16-bit segmentation table in the multiway range search is 16 bits. One bit is used to distinguish between information and node pointers. Therefore, the information array can have at most 32K elements. It means that the multiway range search scheme proposed in [12] can only support no more than 32K routing entries which is smaller than that of the current backbone routers. To remedy this drawback, an entry larger than 16 bits must be used, and the required memory increases. Therefore, the proposed multiway prefix search is better than the multiway range search in this aspect.

7. Conclusions

In this paper, we introduced a new method to compare prefixes of different lengths. By inserting some auxiliary prefixes, a sequential list of prefixes can be constructed and searched using binary search. The proposed scheme uses only 32 bits to encode one prefix. We showed that the maximum number of total prefixes generated is less than $2N$, where N is the number of original prefixes. The sequential list can be easily extended to a multiway search structure and be enhanced by a 16-bit segmentation table. By considering the CPU cache of 32 and 64-byte cache lines, our scheme can achieve the worst-case numbers of memory accesses which are 4 and 3, respectively. The performance analyses and measurements using both IPv4 and IPv6 tables showed that the proposed prefix search schemes perform better than the existing lookup schemes both in terms of lookup latency and memory consumption.

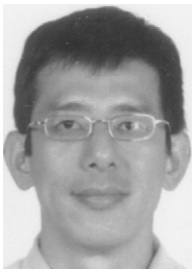
Acknowledgements

We would like to express our sincere thanks to the editors and the reviewers, who gave very insightful and encouraging comments. This work was supported in part by the National Science Council, Republic of China, under Grant NSC-93-2213-E-006-085.

References

[1] A. Basu, G. Narlikar, Fast incremental updates for pipelined forwarding engines, in: Proceedings of INFOCOM 2003.
[2] CERNET BGP VIEW Global Internet, http://bgpview.6test.edu.cn/.
[3] Y. Chang, A small IP forwarding table using hashing, IEICE Transactions on Communication E88-B (1) (2005) 239–246.
[4] T. Chiueh, P. Pradhan, High performance IP routing table lookup using CPU caching, in: INFOCOM, 1999, pp. 1421–1428.
[5] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: ACM SIGCOMM, 1997, pp. 3–14.
[6] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, in: INFOCOM, 1998, pp. 1240–1247.
[7] N.F. Huang, S.M. Zhao, J.Y. Pan, C. A.Su, A fast IP routing lookup scheme for gigabit switching routers, in: Proc. INFOCOM, 1999.
[8] G. Huston, Analysis of the Internet's BGP routing table, Internet Protocol Journal 4 (1) (2001).
[9] M. Akhbarizadeh, M. Nourani, IP routing based on partitioned lookup table, in: Proceedings of the IEEE International Conference on Communications (ICC), April 2002, pp. 2263–2267.
[10] Intel Corporation, IA-32 Intel Architecture Software Developer Manual Volume 2 – Instruction Set Reference, 2003.
[11] C. Labovitz, G. Malan, F. Jahanian, Internet routing instability, in: Proceedings of ACM Sigcomm, September 1997, pp. 115–126.
[12] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Transactions on Networking 3 (3) (1999) 324–334.

[13] H. Lu, S. Sahni, Enhanced interval trees for dynamic IP router-tables, IEEE Transactions on Computers 53 (12) (2004) 1615–1628.
[14] H. Lu, S. Sahni, O(log$n$) dynamic router-tables for prefixes and ranges, IEEE Transactions on Computers 53 (10) (2004) 1217–1230.
[15] D. Meyer, University of Oregon Route Views Archive Project: oix-damp-snapshot-2002-12-01-0000.dat.gz, http://archive.routeviews.org/.
[16] S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, IEEE Journal on Selected Areas in Communications 17 (6) (1999) 1083–1092.
[17] M.A. Ruiz-Sanchez, E.W. Biersack, W. Dabbous, Survey and taxonomy of IP address lookup algorithms, IEEE Network Magazine 15 (2) (2001) 8–23.
[18] S. Sahni, K.S. Kim, Efficient construction of multibit tries for IP lookup, IEEE/ACM Transactions on Networking 11 (4) (2003) 650–662.
[19] S. Sahni, K. Kim, O(log$n$) dynamic router-table design, IEEE Transactions on Computers 53 (3) (2004) 351–363.
[20] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Transactions on Computer Systems 17 (1) (1999) 1–40.
[21] X. Sun, Y.Q. Zhao, An on-chip IP address lookup algorithm, IEEE Transactions on Computers 54 (7) (2005).
[22] S. Suri, G. Varghese, P.R. Warkhede, Multiway range trees: scalable IP lookup with fast updates, in: Global Com 2001.
[23] N. Yazdani, P.S. Min, Fast and scalable schemes for the IP address lookup problem, in: Proceedings of IEEE High Performance Switching and Routing 2000.
[24] F. Zane, G. Narlikar, A. Basu, CoolCAMs: Power-efficient TCAMs for forwarding engines, in: Proceedings of IEEE INFOCOM, 2003.
[25] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, High Speed Scalable IP Lookups, in: Proceedings of ACM SIGCOMM'97, Cannes, France, September 1997.
[26] F. Baker, Requirements for IP Version 4 Routers, RFC 1812, June 1995.



Yeim-Kuan Chang received the MSc degree in computer science from University of Houston at Clear Lake in 1990 and the PhD degree in computer science from Texas A&M University, College Station, Texas, in 1995. He is currently a assistant professor in the department of computer science and information engineering at National Cheng Kung University, Tainan, Taiwan, Republic of China. His research interests are in the areas of computer architecture, parallel processing, Internet technology, and computer networking.