High-performance IP forwarding with efficient routing-table update

Chia-Tai Chan\textsuperscript{a,1}, Pi-Chung Wang\textsuperscript{a,*, 1}, Shuo-Cheng Hu\textsuperscript{b,2}, Chung-Liang Lee\textsuperscript{a,1}, Rong-Chang Chen\textsuperscript{c,3}

\textsuperscript{a}Telecommunication Laboratories, Chunghwa Telecom Co., Ltd. 7F, No.11 Lane 74 Hsin Rd. Sec. 4, Taipei 106, Taiwan, ROC
\textsuperscript{b}Department of Information Management, Ming-Hsin University of Science and Technology, Hsinchu 304, Taiwan, ROC
\textsuperscript{c}Department of Logistics Engineering and Management, National Taichung Institute of Technology, Taichung 404, Taiwan, ROC

Received 29 April 2002; revised 24 January 2003; accepted 26 January 2003

Abstract

There has been an extensive study in constructing the routing tables during the past few years. Although the existing works have certain advantages, those approaches either use complicated data structures which result in large storage requirement and high complexity for updating/building the forwarding table or they are not scalable to fit in Internet protocol version 6 (IPv6). In this work, we propose a fast forwarding-table construction algorithm. With the modified multiway search tree, we can further reduce the depth of the tree and eliminate the storage for pointers. It leads to reduce the FT size and shorten the routing-table lookup time. While considering the route flaps, the forwarding performance will degrade by only 3.1\% with 4000 BGP updates per 30 s in the worst case. Moreover, it is simple enough to fulfill the need of the fast packet forwarding. An extension approach to solve the IPv6 routing lookup is also presented for the future deployment.

\textcopyright 2003 Elsevier Science B.V. All rights reserved.

Keywords: Internet protocol version 6; Routing-table; Fast forwarding-table construction algorithm

1. Introduction

Two important performance metrics of routing algorithm are the packet-forwarding rate and the routing-table size. The routing-table update rate is less obvious but just as important to router performance. Since the router cannot keep up with all of the updates may trigger a condition referred to as route flaps. It significantly causes poor network performance and degrades the overall efficiency of the Internet infrastructure. Speeding up the packet forwarding in the Internet backbone requires high-speed transmission facilities and high performance routers. The transmission technology keeps evolving and provision of multi-gigabit fiber links is commonly available. Consequently, the key to increase the capacity of the Internet lies in the fast router [7]. A high-performance router must have enough internal bandwidth to switch packets between its interfaces at multi-gigabit rates and enough packet processing power to forward multiple millions of packets per second (MPPS) [8]. Switching in the router has been studied extensively and solutions for fast packet processing are deployed commercially. As a result, the remaining obstacle for the high performance router design is the relatively slow IP lookup scheme. A router must search forwarding tables (FT) using the destination address (DA) as the key, and determine which table entry represents the best route to forward the packet to its destination. By the deploying the CIDR in 1993 [9], IP routes have been identified by a (route prefix, prefix length) pair, where the prefix length varies from 1 to 32 bits. Due to the fact that table entries have variable lengths and that multiple entries may represent the valid routes to the same destination, the search of best match prefix (BMP) may be time consuming, especially in a backbone router with a large number of table entries.

\textsuperscript{*} Corresponding author. Tel.: +886-237078697; fax: +886-22345700.
E-mail addresses: abu@cht.com.tw (P.-C. Wang), ctchan@cht.com.tw (C.-T. Chan), schu@mis.mhit.edu.tw (S.-C. Hu), chlllee@cht.com.tw (C.-L. Lee), rcchens@ntit.edu.tw (R.-C. Chen).
\textsuperscript{1} Tel.: +886-2-23265631; fax: +886-2-23445700.
\textsuperscript{2} Tel.: +886-3-5593142x3442.
\textsuperscript{3} Tel.: +886-4-2221-1181x2298.
operation. By combining the sorting with the stack operation, the proposed algorithm can finish the table construction much faster than that of BSP. Moreover, we reorganize the data structure of the multiway search tree. With the refined design, the pointer storage is removed and the number of ways in each node is increased, this results in less depth. We evaluate the performance of our proposed algorithm based on routing-table update rate and average/worst case forwarding rate. Experimental results show that it handles more than 15,000 RUPS and forwards more than 10 MPPS on average, this indicates that our proposed algorithm outperforms the BSP. While considering the route flaps, the forwarding performance will degrade by only 3.1% with 4000 BGP updates per 30 s in the worst case. An extension approach to solve the Internet protocol version 6 (IPv6) [1] routing lookup is also presented for the future deployment.

The rest of this article is organized as follows. Section 2 gives a brief discussion of previous works. In Section 3, we present the proposed FT construction algorithm. The performance evaluation is addressed in Section 4. The enhanced approach to IPv6 routing lookup is introduced in Section 5. Section 6 concludes the work.

2. Related works

There has been a remarkable interest in the organization of routing tables (RT) during the past few years. The proposals can be divided into protocol, hardware and software solutions.

2.1. Protocol based solutions

One possible way to deal with the bottleneck of IP lookup process is to insert extra information (i.e. fixed label) into the packet header to speed up the packet forwarding at routers. Since label switching is based on the exact match of short label, it is easier to build a high-speed router [18]. The typical proposal are IP Switching, Tag Switching [16,17]. IP Switching and Tag Switching bind the routing information of flow’s packet to an ATM virtual channel identifications (VCIs) and threaded indices, respectively, to build FT. However, IP routing lookup is inevitable to perform the forwarding decision in the binding process.

2.2. Hardware based solutions

In the past few years, designers use caching to achieve high-speed IP routing lookups. The caching approach is problematic, such as potentially diluting the performance of caching with hundreds of addresses that map to the same prefix. Moreover, the future backbone routers are expected to forward packets at tera-bit rates and have to maintain hundreds of thousands of prefixes. Although studies have shown that caching in the backbone routers can result in hit ratios up to more than 90% [19,20], the large, fully associative caches (content-addressable memory, CAM) are usually expensive and not scalable.

Gupta et al. [4] presented fast routing-lookup schemes based on a huge DRAM. The scheme accomplishes a maximum of two memory accesses for a lookup in a FT of 33 Mb, as shown in Fig. 1. By adding an intermediate-length table, the FT can be reduced to 9 Mb; however, the maximum number of memory accesses for a lookup is increased to three. When implemented in a hardware pipeline, it can achieve one route lookup every memory access. This furnishes about 20 MPPS. Huang et al. [13] further improve it by fitting the FT into SRAM. In Ref. [15], the authors adapted the properties of RT and proposed two novel entry-compression schemes. By resolving the structural hazard of memory, the algorithm can be implemented with pipelined hardware easily. Basically, these schemes cannot support incremental update, thus incur high update cost.

![Fig. 1. Indirect-lookup mechanism.](image)
2.3. Software based solutions

Regarding software solutions, the schemes can be classified as tree-based algorithms and binary search on Hash-tables/prefixes and described as follows.

2.3.1. Tree based algorithm

The trie is a general-purpose data structure for storing strings. The idea is very simple: each string represents the corresponding path from the root to the leaf in the trie. This simple structure is not very efficient. The number of nodes may be large and the average depth (the average length of a path from the root to a leaf) may be long. The traditional technique to overcome this problem is to use path compression when each internal node with only one child is removed. It has to record those removed nodes somehow. A simple technique is to store a useful information in each node, called the skip value, which indicates how many bits have been skipped on the path. A path-compressed binary trie is sometimes referred to as a Patricia trie, which is implemented in the IP lookup of NetBSD. Consider the example shown in Fig. 2. There are five bit streams in the left part. Based on the given RT, the constructed Patricia trie is shown in the right part of Fig. 2. Obviously, the path compression is to compress the parts of the trie that are sparsely populated.

Level compression is a recently introduced technique for compressing parts of the trie that are densely populated. The idea is to replace the \( i \) highest complete levels of the binary trie with a single node of degree \( 2^i \); this replacement is performed recursively on each subtrie. The level-compressed version of the trie in Fig. 2 is shown in Fig. 3. In Fig. 3, one might find that there are eight leaves generated but only five route prefixes in the table. Accordingly, three nodes are over-generated in the trie, which is noted in Fig. 3.

The performance of the LC-trie is much better than that of the Patricia trie. Since the comparison of string lengths is based on the machine word size and it is more efficient to compare more bits at a time in order to reduce the number of comparisons and memory accesses. However, the implementation in Ref. [12] uses array representation by laying out the LC-trie nodes in breadth first order (first the root, then all the trie nodes at the second level from left to right, then third level nodes etc.). Each node only carries the first address of its children. While performing address lookup, the trie search does not stop until the leaf node is reached. The fetched node may not be the BMP due to the path compression. If the prefix in the node does not match the address, the shorter prefix of the node will be tracked to find the BMP. However, the array layout and the requirement for full subtries make updates slow in the worst case. In addition, it might cause almost every element in the array representation to be moved. Thus while LC-trie have reasonable average lookup speeds, they feature long insertion time and worst-case lookup performance.

The data structure presented in Ref. [3] was similar to LC-trie [12]. It is also a binary trie structure, and it allows multway branching. By using a standard trie representation with arrays of children pointers, insertions and deletions of prefixes can be supported. However, to minimize the size of the initial trie, complex dynamic programming is used. Degermark et al. [2] proposed a trie-like data structure. The Lulea scheme compresses multibit trie nodes so that the entire data structure can be placed in SRAM. In contrast with the Patricia/LC tries to compresses the trie structurally, Lulea scheme compresses the information in a node. The nodes are compressed by representing repeated elements in a node array only once. By using a bitmap to specify the number of repeated times of an element, as shown in Fig. 4. With software implementation, the minimum and maximum number of memory accesses for a lookup are two and twelve, respectively.

To speed up the counting of set bits, the algorithm accompanies each bitmap with a summary array. It contains
a cumulative count of the number of set bits associated with the fixed size chunks of the bitmap. Using this structure, it is able to compact a large RT with 40,000 entries into a table with 150–160 Kb size. However, it has two disadvantages. First, counting bits requires at least one extra memory reference per trie node. Second, the implicit use of leaf pushing leads to a large worst-case insertion time and the insertion of a prefix into the root node entry causes the information to be pushed to thousands of leaf nodes.

2.3.2. Binary search on hash-tables/prefixes

Waldvogel et al. [10] took a different approach by using binary search on hash-tables. The routing prefixes are stored in the hash-tables according to their lengths. A lookup is performed as a binary search within the different tables, thus one hash-table lookup is performed at each step of IP lookup. This is potentially expensive, but works well in practice for a number of RTs. Since the lengths of the entries are typically unevenly distributed, the search starts with the most probable prefix length. For the 128 bit addresses of IPv6, this approach may require as many as seven hash-table lookups. Each lookup might in turn require several memory accesses. Also, to guide the binary search to the correct hash table, the marker with the associated BMP has to be inserted. This might not only increase the number of entries, but also raise the table update complexity.

In Ref. [11], the software-based binary search was applied to the routing prefixes directly, rather than the hash-tables. Through pre-computation, the relationship between the routing prefixes and the IP address space can be calculated. As a result, searching the suitable region performs the IP routing lookup. This performance can be further improved by employing the cache line alignment. For a database of \( N \) prefixes with address length \( W \), the native binary search scheme needs \( O(W \times \log N) \) searches. This improved scheme takes only \( O(W + \log N) \) searches. Through experiments, it can achieve more than 2 MPPS worst-case performance using a PC with 200 MHz CPU. This scheme is attractive, even for IPv6, because of its bounded memory requirement. However, the prefix insertion/deletion will result in table reconstruction due to the recalculation of pre-computed information. It achieves the fast forwarding at the cost of the increasing complexity of prefix insertion process. For example, in the worst case, the table construction may take as long as 5.8 s. For prefixes with length smaller than 16 bits, the worst-case update time is about 1.25 ms, while the estimated worst and average update time for prefixes with length larger than 16 bits are 352 and 20 ms, respectively, [11]. Obviously, the RT construction is not fast enough to handle the rapid route update, i.e. 100 RUPS, in a backbone router [5]. According to the brief description in Ref. [11], the table update procedure should include entry duplication, sorting and stack operation. Also, the required storage is too large to utilize the L2 cache.

Although the existing works feature certain advantages, however, those approaches either use complicated data structures which result in high complexity for updating/building the FT, such as Refs. [3,10,11], or they are not scalable to fit in IPv6 [2,4,12,15].

3. The proposed IP forwarding table construction algorithm

Before we introduce the proposed algorithm, the conceptual architecture of the IP router is shown in Fig. 5. With the advent of the high-speed switch capacity and increased traffic volume, distributed routing architecture has become a significant improvement to achieving high capacity in router design. It mainly consists of a network processor and forwarding engine. The network processor executes the routing protocols, such as BGP and OSPF, and maintains a RT. In each line card, the forwarding engine employs a FT to make the routing decision for packet forwarding. The FT is derived from the RT and contains an index of IP prefix associated with an outgoing interface. Once the route table exchanges, the forwarding engine will update its FT based on the information from the network processor. This separation is used to ensure the routing instability does not impact the performance of packet forwarding engine. While route updating, the forwarding engine...
3.1. Routing prefix pre-computation

Let \( S_i, E_i \) and \( l_i \) be the starting address, ending address and the length of the route prefix \( P_i \), respectively. With the sorted prefixes, we can accomplish the prefix pre-computation easily by just processing each route prefix sequentially. According to the routing interval concept, a routing prefix \( P_i \) implies an interval that can be identified by \((S_i, P_j, P_k)\) and \((E_i, P_j, P_k)\). The starting and ending address are just appending \((32 - l_i) 0s\) and \((32 - l_i) 1s\), respectively. The second element \( P_j (P_k) \) of the ordered entry \((S_i, P_j, P_k)\) represents a prefix value equals \( S_i (E_i) \). The third element \( P_j (P_k) \) of the ordered entry \((S_i, P_j, P_k)\) represents a prefix value which is greater than \( S_i (E_i) \). Initially, the value of \( P_j \) is unknown since it might refer to a prefix processed later. Assume \( S_1 < S_2 < \cdots < S_N \) \((N\) is the number of route prefixes, \(1 \leq i \leq N\)). If \( S_i = S_j \), then \( i < j \) if and only if \( l_i < l_j \).

For the route prefix \( P_j \); both values of ‘equal’ (i.e. the second element) and ‘large’ (i.e. the third element) fields following starting and ending address in the order entry can be initially filled with \( P_j \) and \( P_j \), \( P_j \), respectively. However, it is an imprecise interval since it overlaps with other interval in the boundary that causes the ambiguous routing lookup. The routing prefix pre-computation is used to distinguish the intervals, both values will be overwritten with the correct prefix identifier. We adopt two arrays, \( L_1 \) and \( L_2 \), for the routing prefix pre-computation. The first one is used to store the final generated entry, while the other is implemented as a stack which is use to perform ‘push’ and ‘pop’ operations. To simply the description, we use the subscript \( top \) and \( rear \) to represent both top and rear intervals in \( L_1 \) and \( L_2 \), respectively. Initially, the first entry \((S_0, P_0, P_0)\) with starting address will be inserted into \( L_1 \) directly, while the associated entry \((E_0, P_0, null)\) with ending address will be pushed into \( L_2 \). To place the following entry \((S_i, P_j, P_k)\) with starting address in the correct position, we check whether the address of the top entry in \( L_2 \) is smaller than \( S_i \). If so, we pop the top element of \( L_2 \) and append it to \( L_1 \). We repeat these steps until there is no entry smaller than \( S_i \). When the next routing prefix \((S_i, P_j, P_k)\) is processed, the interval is further partitioned into three regions, \(140.113.3.0/255.255.255.0/NH_1\), \(140.113.3.0/255.255.255.0/NH_2\) with length 16, 24, respectively. First, the routing prefix \((S_i, P_j, P_k)\) is used to store the final generated entry, while the other is occupied by a shorter prefix that forms the ‘large’ field is occupied by a shorter prefix that forms the

1. Reducing routing-table space. If we can reduce routing-table size to fit into the high speed SRAM, the performance can be promoted significantly.

2. Fast routing-table reconstruction. Basically, the update cost of the trie-base algorithm is smallest. Other schemes, such as cache, hash and binary search, might potentially require full table refresh. To avoid this problem, one usually adopted the 16 bits pre-computation table. By dividing the whole address space into 64 K small space, the problem can be alleviated.

3. Simplicity. It should make the data structure and the routing lookup operation as simple as possible, such as using pre-computation to avoid search complexity shown in Ref. [11].

In Ref. [14], we have demonstrated that most of routing prefixes are related to neighbor clouds and the routing prefixes are simple and much fewer for the remote cloud. We call this phenomenon as routing locality and propose a new routing concept named as ‘Routing Interval’. By sorting the routing prefixes based on their lengths, we can build a new next-hop array in which each element maps to an IP address interval and is filled with related next-hop. For example, we consider two routing prefixes \(140.113.0.0/255.255.0.0/NH_1 \) and \(140.113.3.0/255.255.255.0/NH_2\) with length 16, 24, respectively. First, the routing prefix \((140.113.0.0/255.255.0.0/NH_1)\) creates an IP address interval \(140.113.0.0, 140.113.255.255/NH_1\). When the next routing prefix \((140.113.3.0/255.255.255.0/NH_2)\) is processed, the interval is further partitioned into three regions, \(140.113.3.0, 140.113.2.255/NH_1, 140.113.3.0, 140.113.3.3.255/NH_2\), \(140.113.4.0, 140.113.255.255/NH_1\), respectively.

The similar idea is also shown in Ref. [11]. However, as described above, the proposed scheme cannot support fast update that is required in the backbone routers. Consequently, we propose an efficient forwarding-table construction algorithm which consists of prefix pre-computation and enhanced multiway search tree. We also present the detail procedure of the table update.

3.1. Routing prefix pre-computation

Let \( S_i, E_i \) and \( l_i \) be the starting address, ending address and the length of the route prefix \( P_i \), respectively. With the sorted prefixes, we can accomplish the prefix pre-computation easily by just processing each route prefix sequentially. According to the routing interval concept, a routing prefix \( P_i \) implies an interval that can be identified by \((S_i, P_j, P_k)\) and \((E_i, P_j, P_k)\). The starting and ending address are just appending \((32 - l_i) 0s\) and \((32 - l_i) 1s\), respectively. The second element \( P_j (P_k) \) of the ordered entry \((S_i, P_j, P_k)\) represents a prefix value equals \( S_i (E_i) \). The third element \( P_j (P_k) \) of the ordered entry \((S_i, P_j, P_k)\) represents a prefix value which is greater than \( S_i (E_i) \). Initially, the value of \( P_j \) is unknown since it might refer to a prefix processed later. Assume \( S_1 < S_2 < \cdots < S_N \) \((N\) is the number of route prefixes, \(1 \leq i \leq N\)). If \( S_i = S_j \), then \( i < j \) if and only if \( l_i < l_j \).

For the route prefix \( P_j \); both values of ‘equal’ (i.e. the second element) and ‘large’ (i.e. the third element) fields following starting and ending address in the order entry can be initially filled with \( P_j \) and \( P_j \), \( P_j \), respectively. However, it is an imprecise interval since it overlaps with other interval in the boundary that causes the ambiguous routing lookup. The routing prefix pre-computation is used to distinguish the intervals, both values will be overwritten with the correct prefix identifier. We adopt two arrays, \( L_1 \) and \( L_2 \), for the routing prefix pre-computation. The first one is used to store the final generated entry, while the other is implemented as a stack which is use to perform ‘push’ and ‘pop’ operations. To simply the description, we use the subscript \( top \) and \( rear \) to represent both top and rear intervals in \( L_1 \) and \( L_2 \), respectively. Initially, the first entry \((S_0, P_0, P_0)\) with starting address will be inserted into \( L_1 \) directly, while the associated entry \((E_0, P_0, null)\) with ending address will be pushed into \( L_2 \). To place the following entry \((S_i, P_j, P_k)\) with starting address in the correct position, we check whether the address of the top entry in \( L_2 \) is smaller than \( S_i \). If so, we pop the top element of \( L_2 \) and append it to \( L_1 \). We repeat these steps until there is no entry smaller than \( S_i \). When the interval is further partitioned into three regions, \(140.113.3.0, 140.113.2.255/NH_1, 140.113.3.0, 140.113.3.3.255/NH_2\), \(140.113.4.0, 140.113.255.255/NH_1\), respectively.

The similar idea is also shown in Ref. [11]. However, as described above, the proposed scheme cannot support fast update that is required in the backbone routers. Consequently, we propose an efficient forwarding-table construction algorithm which consists of prefix pre-computation and enhanced multiway search tree. We also present the detail procedure of the table update.
Routing Prefixes

P₀ <p₀,i,nh_n>  P₁ <p₁,i,nh_n>  P₂ <p₂,i,nh_n>  P₃ <p₃,i,nh_n>

The Pre-computation Entries

S₀, P₀, P₀  S₁, P₁, P₁  S₂, P₂, P₂  S₃, P₃, P₃

E₀, P₀, null  E₁, P₁, null  E₂, P₂, null  E₃, P₃, null

Fig. 6. The incremental update of the pre-computation entries.

Note that the pre-computation can be performed based on the incremental manner with linked-list. The entries with starting/ending addresses of a newly incoming prefix can be inserted into the existing list by using simple linear search. The entry fields must be revised according to its nearby entries. We use the example shown in Fig. 6 to illustrate the operation of insertion. Assume that the P₂ and P₃ are the inserted prefixes. As the insertion of the P₂ at first, the entry S₂, P₂, P₂ can be inserted straightforward. To process the entry with ending address, the entry (E₂, P₂, P₀) will be inserted where P₀ inherits the same field from the previous entry. In addition, the ‘large’ field of the previous entry will be modified as P₂ since its following region is occupied by P₂. The relationship between the entries can be derived based on the nested enclosures. The insertion of the entries for P₃ is trivial. However, if E₁ equal to E₀, these two entries have to be merged to avoid ambiguous routing. Reasonably, the entry corresponding to longer prefixes will be kept, which is (E₀, P₀, P₀) in this example. The prefix deletion can be done similarly.

3.2. Enhanced multiway search tree

After deriving the searchable entries, the IP lookup can be performed based on the binary search. However, due to the unbalanced distribution of the route prefixes, 16 bit pre-computation table will be used to speed up the binary search, as used in Ref. [11]. If the possible matched prefix for the incoming packet is shorter than 16 bits, the search will stop at the 16 bit pre-computation table. Otherwise, the entry in the table contains a pointer to the searchable structure. Furthermore, the native binary search can be improved with cache line alignment [11]. Since in each memory access it will fetch entire the cache line into L2 cache, multiple keys can be filled into a single cache line. With the multiway search tree, the number of memory accesses can be reduced.

The next-hop id, rather than the prefix id, to indicate the packet destination. One may argue that by recording the prefix id, route change cost can be alleviated. However, the number of prefixes is too large to represent with 2 bytes (i.e. 65,536 prefixes). By contrast, even for the most modern routers, 1 byte is enough to represent the next-hop id (i.e. 256 next-hops). Thus we can reduce the size of each entry to 4 bytes.

Before constructing the multiway search tree, we pack eight entries into a so-called ‘entry pool’ which is 32 bytes. The key of the first entry will be treated as a flag and inserted into the multiway search tree. Since each flag is 2 bytes, a cache line could allocate 15 keys and the address of the first child node (2 bytes) to construct a 16-way search tree, as shown in Fig. 7.

The IP lookup procedure is presented as follows:

1. Index into the 16 bit pre-computation array with the first 16 bits as the DA.
2. If the value is smaller than 256, it is a next-hop identifier. Stop the search. Otherwise, it is a pointer to the multiway search tree.
3. If the first two bytes of the fetched block is nonzero, the block is not a node, and go to Step 4. Otherwise, the block is a leaf and go to Step 5.
4. Binary search will be used to find out the region i which satisfies $K_i < \text{destination address} \leq K_{i+1}(0 \leq i \leq 14)$. Fetch the block from the address (head + i) and go to Step 3.
5. Again, binary search will be used to find out the region i which satisfies $K_i < \text{destination address} \leq K_{i+1}(0 < i \leq 7)$ in the entry pool. If the DA is equal to $K_{i+1}$, then return the next-hop id $N_{i+1}$. Otherwise, return the next-hop id.

In the worst case, three cache line accesses in the modified multiway search tree would be able to accomplish the IP lookup. Also, with reduced pointer storage and simplified next-hop representation, the required memory is much less than that in the previous scheme. In Table 1, we show the complexity required for different software-based schemes. Note that these complexity measures do not discuss anything about the absolute speed or memory usage.

3.3. Route update

We have discussed the incremental update of the entry pre-computation in Section 3.1. In the following, we deal with the issues raised in the multiway search tree. The procedures of the route update are divided into prefix insertion, deletion and route change. In the following, we address the procedures, respectively.

- Prefix insertion

While a new prefix is inserted, it is always necessary to rebuild the multiway search tree.
Unlike the prefix insertion, we can batch the reconstruction until another prefix insertion. It must correct the multiway search tree by changing the route of the deleted prefix to its shorter prefix. Firstly, we track the sorted routing prefix and record the shorter prefix of the deleted one. Once the deleted prefix is found, we keep on tracking its longer prefixes. Consequently, regions that are not covered by the longer prefixes will be corrected. For each such region, we pick one occupied IP address and use it to traverse the multiway search tree. Then we change the route of the destination entry to the route of its shorter prefix. The detailed algorithm is shown in Fig. 8.

We use a simple example to explain it. As shown in Fig. 9, there are three routing prefixes: \( P_1 \) (210.68/16/NH1), \( P_2 \) (210.68.224/20/NH2) and \( P_3 \) (210.68.243/24/NH3). These prefixes will be mapped into six entries. Once \( P_2 \) is deleted, the related blocks of IP addresses whose route is based on \( P_2 \) will be corrected. Thus, both fields of \( E_2 \), the ‘large’ field of \( E_4 \) and the ‘equal’ field of \( E_5 \) will be changed to NH1. Consequently, to modify the entries, the associated IP addresses will be used to correct the branches in the multiway search tree.

The procedure of route change is akin to that of prefix insertion and prefix deletion. By calculating the related entries from sorted prefixes and change their route in the multiway search tree, the procedure of route change can be done.

### 3.4. Implementation issues

One of the strength of the proposed architecture is its cost-efficiency. As compared to the existing schemes, the proposed scheme could achieve fairly good performance even with cheap DRAM. The scheme is suitable for software implementation since the lookup procedure is simple. While implementing in hardware, the performance can be further improved by adopting fast SRAM and wide memory bus.

The need for memory management is a major obstacle for the implementation. In our data structure, we allocate the tree node contiguously. This might cause external fragmentation [21]. This shortcoming is also incurred by most software-based algorithms. This is because the software-based algorithms usually use flexible data structure to improve the performance, thus results in variable-size allocation. Since we have optimized the proposed scheme for cheap DRAM, the memory-management could be simplified by adopting large enough memory. For example, an extremely large 128 Mb DRAM chip for the proposed scheme costs less than US $5 currently. Thus the implementation of the proposed scheme is relatively simple compared to the other software-based algorithms.

### 4. Performance evaluation

To simplify the comparison with multiway search, we need a compatible platform as used in previous work [11].

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Build</th>
<th>Search</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patricia trie</td>
<td>( O(NW) )</td>
<td>( O(W) )</td>
<td>( O(NW) )</td>
</tr>
<tr>
<td>Binary search on hash table</td>
<td>( O(N \log W) )</td>
<td>( O(\log W) )</td>
<td>( O(N \log W) )</td>
</tr>
<tr>
<td>Multibit trie</td>
<td>( O(hW^2) )</td>
<td>( O(h) )</td>
<td>( O(hW) )</td>
</tr>
<tr>
<td>LC tries</td>
<td>( O(hN) )</td>
<td>( O(h) )</td>
<td>( O(hN) )</td>
</tr>
<tr>
<td>Multiway search tree</td>
<td>( O(N) )</td>
<td>( O(\log_{32}(2N)) )</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>Proposed scheme</td>
<td>( O(N) )</td>
<td>( O(\log_{32}(2N)) )</td>
<td>( O(N) )</td>
</tr>
</tbody>
</table>

\( N \): the number of the prefixes, \( W \): the length of the address, \( h \): the height of the trie.
We choose a 300 MHz Pentium II that has a 512 Kb L2 cache and runs Windows NT for the experiment. Five RTs obtained from the IPMA project [6] on July 27, 2000 are used. We will show the performance of the proposed scheme with respect to three metrics: worst/average search time, storage and construct/update time.

To show the realistic performance of the proposed algorithm, the simulation should avoid any L1 cache hit. For example, the result derived based on continuous address lookup will be improved externally since each required data might be located in L1 cache. To avoid this problem, we should adopt a larger working space. For example, to evaluate the worst-case lookup performance, we choose the segment (first 16 bits of IP address) with a maximum number of route prefixes (about 130). The required storage for this segment is 1216 bytes after memory bus alignment. Then we make 1024 copies of this segment and construct a FT for them. The total storage for building the FT is larger than 1.2 MB, which is much larger than the size of L1 cache (32 Kb). Consequently, we perform IP lookup for the address with the maximum number of memory accesses to these 1024 segments. This will ensure that each acquired block is fetched from the main memory, this reflects the real situation. The worst-case lookup time can be calculated by dividing the total elapsed time with the number of iterations (1024). This method will be deployed in our experiment for a realistic result.

The required storage for different RT is shown in Fig. 10. The numerical results of the BSP are also listed to demonstrate the improvement. For the most RTs, the required storage of the proposed scheme is smaller than that of BSP even with much more prefixes in the used tables. This is mainly because the proposed multiway search tree can further utilize the memory. By saving the pointer storage and deploying the next-hop representation, the memory efficiency is increased significantly.

The build/update time of the experimental results are shown in Fig. 11. Note that we can only compare the build time since the update time is not available in Ref. [11]. Obviously, the build time of the proposed scheme is much less than that of BSP. There are two reasons: (1) the efficient construction algorithm largely reduce the computation complexity. Furthermore, through linked-list implementation, the memory movements are minimized. (2) The required storage is much smaller with the compressed data structure, thus increases the cache hit-ratio. To measure the worst-case update time, we use the prefix insertion time since it will cause the reconstruction of the searchable entries and the multiway search tree. In Ref. [11], in which authors claimed that the estimated average and worst case update time are 20 and 350 ms, respectively. With our algorithm, the worst-case update time is less than 230 ms. The average update time (about 60 ms) can be derived by dividing the total build time
(188 ms) with the number of multiway search trees (about 3000). As a result, the proposed scheme can support more than 15,000 RUPS on average. It also means that the forwarding performance will degrade by only 0.8% and 3.1% with 4000 BGP updates per 30 s in the average and worst case, respectively.

The worst-case/average lookup time is shown in Fig. 12. For the same reason as described, we only compare the worst-case lookup time of BSP. Apparently, the worst-case lookup performance could illustrate the effect of the new data structure. By adopting more branches in the multiway search tree, the lookup performance is improved effectively. Even with the largest RT Mae-East (58,000 entries), the lookup time is still less than the lookup time of the BSP in the 38 K-entry table. Note that as the number of routing prefixes increasing, the speed improvement of the proposed scheme will be more notable since the large number of branches will reduce the tree depth significantly.

We further examine the average performance. By assigning each IP address with same weight, the average performance is calculated by taking the total elapsed time and dividing it by the number of IP addresses ($2^{32}$). As shown in Fig. 12, the proposed scheme can achieve more than 10 MPPS. Note that the performance can be further improved by using faster memory or wider CPU cache line.

In Fig. 13, we further compare the proposed scheme with the other existing algorithms. We also scaled all results of previous works to 300 MHz CPU to ease the comparison. While considering the lookup speed, only the multibit trie offered better performance. However, the proposed scheme features smaller storage and the multi-bit trie needs complex dynamic programming to
minimize the size of the tree, thus features much slower update time (2.5 ms). Also, the proposed scheme could offer better storage scalability with respect to the number of prefixes. The rest schemes, including LC tries, binary search on hash tables, and Lulea compressed tries, all potentially require changing the complete data structure for a route update, thus they are likely to feature slow insertion/deletion times.

5. Extension approach to IPv6 routing lookups

After demonstrating the performance of the proposed scheme, we further present how to extend the enhanced multiway search tree to support IPv6 routing lookups. The extension approach provides flexible characteristics for unpredictable routing behavior. In the following, we only describe the essential parts for IPv6; the overlapped information will be omitted here.

5.1. Routing prefix pre-computation

To cope with IPv6 address, the 128 bits should be cut into multiple chunks, such as 32 or 64 bit. Based on the designated chunk size, the routing prefixes can be sorted by multiple stages. After sorting the \(i\)th chunks, the \((i + 1)\)th chunk will be sorted based on the sorting results of 1 \(- i\) chunks. With the sorted prefixes, the prefix pre-computation can be done with the same algorithm, as described in Section 3.1. The sorted prefixes and the pre-computed entries are maintained with linked-list, thus the incremental insertion/deletion is also supported.
5.2. Enhanced multiway search tree coupling with dynamic chunk selection

Unlike the routing lookup for IPv4 packet, the much longer address in IPv6 makes the 16 bit pre-computation table useless. Thus, we use only one multiway search tree to perform IP routing lookup. To eliminate the update cost, the proposed algorithm restricts the updated data only in the corresponding subtree. The unit of the tree construction procedure is based on the chunk. To begin with, the multiway search tree for the first chunk is built. For the entries with the identical subprefix in the first chunk will be constructed based on their subprefix in the second chunk and so on. The basic data structure is unchanged, however, the chunk size can be variable according to the density of the entries. With a larger chunk, the number of keys in each cache line is fewer, thus results in less branches. For the chunks with few entries, a larger chunk (i.e. 64 bits) is chose to bypass more bits in one cache line access. Otherwise, a 32 or 16 bit chunk is used to increase the number of branches. The effect of the different chunk size is shown in Fig. 14. Assume that the number of branches is four with one-word chunk and two with two-word chunk. Each dotted-line square corresponds to one node. The multiway search tree with fixed one-word chunk is shown in the left side. After changing the second chunk to two words, the height of subtrees of B and C are reduced due to their sub-strings can be fit into single node. However, the counter-effect is resulted in the subtree of A since there are seven sub-strings which need four nodes with increased height by one to accommodate. Thus the subtree of A should remain unchanged. Note that the size of the subsequent chunks may not be the same to achieve optimal selection.

The hierarchical construction based on the chunks may eliminate the possible variations caused by the route updates. The detailed steps are omitted. The search procedure is akin to the basic algorithm. To measure the possible performance, we assume the chunk with 16 and 128 byte cache line are used. In the worst-case (128/16 = 8) cache-line accesses is required with 64-way search tree and supports at most 64^8 (= 2^{18}) routing prefixes. The lookup speed can be further improved by deploying larger cache line.

6. Conclusion

In this work, we aim at enhancing the BSP for prefixes longer than 16 bits with a fast forwarding-table construction algorithm. Since the update cost ties to the table construction process, we addressed the incremental update for the pre-computation and proposed a fast forwarding-table construction algorithm. We also resolved the ambiguous lookup problem caused by duplicate entry. With the reorganized data structure, the number of the branches is increased to lessen the height of tree. Our algorithm can achieve 10 MPPS and support more than 15,000 RUPS with the current 58,000-entries RT in a backbone router. While considering the route flaps, the forwarding performance will degrade by only 3.1% with 4000 BGP updates per 30 s in the worst case. It is obvious that the proposed algorithm improves the router performance significantly.

For the future deployment of IPv6, we introduced the modified multiway search tree with different chunk size. The proposed data structure is flexible to match up various RTs. The estimated search time is under eight cache-line accesses. Certainly, it could feature higher performance by using processors with larger cache line.

References


