Computer Networks 106 (2016) 77–90

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Compressing IP Forwarding Tables with Small Bounded Update Time

Yuanyuan Zhang[a], Mingwei Xu[a,*], Ning Wang[b], Jun Li[c], Penghan Chen[d], Fei Liang[d]

[a] Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, China
[b] Institute for Communication Systems, University of Surrey, United Kingdom
[c] Department of Computer and Information Science, University of Oregon, United States
[d] Broadband Network Research Center, Beijing University of Posts and Telecommunications, China

ARTICLE INFO

Article history:
Received 30 September 2015
Revised 8 June 2016
Accepted 14 June 2016
Available online 20 June 2016

Keywords:
FIB Compression
FIB update
IP address lookup
Tries
Longest prefix matching

ABSTRACT

With the fast development of the Internet, the size of Forwarding Information Base (FIB) maintained at backbone routers is experiencing an exponential growth, making the storage support and lookup process of FIBs a severe challenge. One effective way to address the challenge is FIB compression, and various solutions have been proposed in the literature. The main shortcoming of FIB compression is the overhead of updating the compressed FIB when routing update messages arrive. Only when the update time of FIB compression algorithms is small bounded can the probability of packet loss incurred by FIB compression operations during update be completely avoided. However, no prior FIB compression algorithm can achieve small bounded worst case update time, and hence a mature solution with complete avoidance of packet loss is still yet to be identified. To address this issue, we propose the Unite and Split (US) compression algorithm to enable fast update with controlled worst case update time. Further, we use the US algorithm to improve the performance of a number of classic software and hardware lookup algorithms. Simulation results show that the average update speed of the US algorithm is a little faster than that of the binary trie without any compression, while prior compression algorithms inevitably seriously degrade the update performance. After applying the US algorithm, the evaluated lookup algorithms exhibit significantly smaller on-chip memory consumption with little additional update overhead.

1. Introduction

1.1. Background and motivation

The size of Forwarding Information Bases (FIBs) of backbone routers in the Internet has been increasing by around 15% every year [1]. The FIB (DFZ entries) size exceeded 512K on August 13th in 2014, exceeding the hardware capacity of many legacy Cisco routers [2]. In addition, there were VPN routes which could be as many as the DFZ entries. As a result, it took about a week for these routers to upgrade their hardware capacity, and it has already been observed that the web browsing and content downloading speed was slowed down during the period. In the literature, technical schemes have already been proposed to solve such a problem, and among them FIB compression is a promising way to alleviate the growth pressure of FIBs in the Internet.

In fact, even if the FIB size does not exceed the capacity of routers, FIB compression is still beneficial for IP lookup. Generally, there are two kinds of IP lookup solutions. The first kind is hardware-based solutions, such as TCAM-based solutions [3–6] and FPGA-based solutions [7–10]. For this kind of IP lookup solutions, compressing the FIBs can significantly save hardware cost and power consumption. The second kind is software-based solutions, such as [11–14]. For this kind of IP lookup solutions, compressing the FIBs reduces the probability of cache misses, and thereby achieves faster lookup speed.

As mentioned in BS [15], EAR [16], and FIFA [17], packet loss may happen when the compression or update algorithm is too slow. The FIB after compression is stored and looked up in the data plane of a router. When a FIB update message arrives, the router has to suspend the lookup process and buffer the incoming packets in a queue. The queue can only buffer finite packets, thus the update of the compressed FIB should be as fast as possible. If the update time is not small enough in the worst case, the buffer in the data plane may overflow and packet loss may happen. This is the main reason why vendors and ISPs are not willing to adopt FIB compression algorithms in real routers. Therefore, this paper targets at a practical FIB compression algorithm with small bounded update time.

[*] Corresponding author.
E-mail addresses: zhyyuan1019@gmail.com (Y. Zhang), xmw@cernet.edu.cn (M. Xu).

http://dx.doi.org/10.1016/j.comnet.2016.06.015
1389-1286/© 2016 Elsevier B.V. All rights reserved.

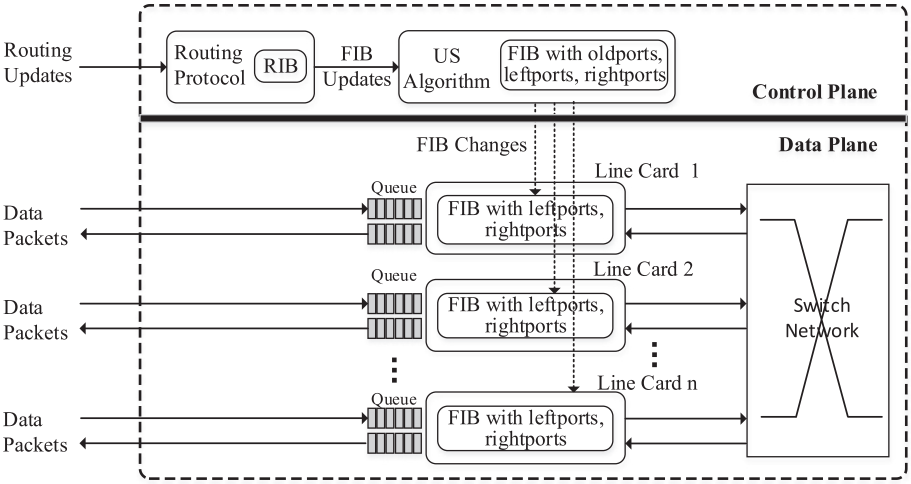


Fig. 1. Router architecture.

1.2. State-of-the-art and their limitations

Due to the significance of FIB compression, various compression solutions have been proposed, such as ORTC [18] and its successors [17,19,20], auto aggregation [21], 4-level [22], entropy compression [23], EAR[16], and NSFIB compression [24], *etc*. Among them, ORTC constructs the optimal FIBs in terms of the number of prefixes. Entropy compression pursues the optimal compression algorithm in terms of information entropy, but the compression results are no longer in the prefix format, thus cannot cooperate with existing IP lookup algorithms. NSFIB is an aggressive compression method which can exceed the optimal compression ratio of ORTC at the cost of changing the forwarding behavior. Although some classic compression algorithms (such as SMALTA [19], EAR, 4-level, auto aggregation, *etc.*) claimed to support fast update, no prior algorithm is able to achieve small bounded worst case update time. *Only when the worst case of update time is small bounded, the risk of packet loss during update can be fundamentally avoided.*

1.3. Proposed solution overview

In this paper, we propose the `Unite and Split (US)` compression algorithm. The top level strategy of conventional FIB compression algorithms is to *either make the best effort for compression ratio or to identify a trade-off between compression ratio and update speed*, but it should be noted that no prior compression algorithm has a reasonable worst case bound of update time. In contrast, the objective of our US algorithm is to *make the best effort for compression ratio in the premise of small bounded update time*.

We use a trie[1] structure to illustrate the key compression technique of our proposed US algorithm. As shown in the first trie of Fig. 2(a), it has three nodes with non-empty next hops: *q and its two child nodes* q_1 *and* q_2. According to the longest prefix matching rule, an IP address either matches q_1 or q_2. In other words, there are at most two lookup results for any incoming IP packet. Therefore, we can replace this trie by one node with two next hops. In other words, the two child nodes q_1 and q_2 can be compressed (`united`) into their parent node q with two next hops, where the left next hop belongs to q_1, and the right next hop belongs to q_2. Similarly, when two of the three nodes have non-empty next hops (the middle three tries in Fig. 2(a)), they can be compressed into one node with two next hops. However, we do not always perform such compression because when only one of the three nodes has a next hop (such as the three tries in Fig. 2(b)), such compression does not reduce the number of prefix nodes[2], but brings additional update overhead. In this case, we `split` the prefix node to guarantee that every node has either two next hops or none for the sake of storage and lookup efficiency. For each trie node, we use the variable *oldport* to store the next hop before compression for the sake of correct update, and use variables *leftport* and *rightport* to store the left next hop and right next hop after compression respectively.

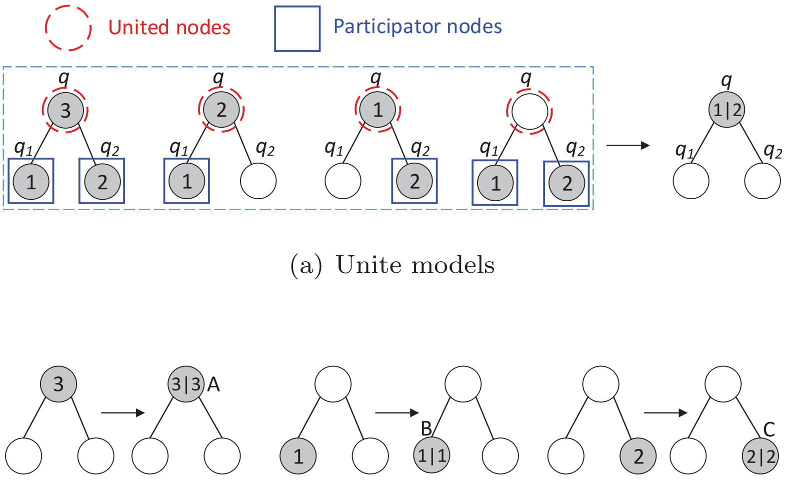
The US algorithm consists of two kinds of operations: unite and split, and it traverses the trie twice. In the first postorder traversal of the trie, we conduct the unite operations to reduce the number of prefix nodes. In the second postorder traversal of the trie, we conduct split operations on those nodes which do not participate in the unite operations. To bound the update time, it is guaranteed that every trie node participates in at most one unite operation, and the nodes modified by each unite operation are confined in two adjacent levels. In this way, *at most 3 trie nodes need to be updated by any update message.* The main advantage of our algorithm is that the update speed is fast and the worst case of update time is small bounded. Simulations using real-world FIBs (around 512K entries) and updates on CPU platform with Intel(R) Core i7-3517U 1.9GHz & 2.4GHz and 8GB RAM show that the update speed of US ranges from 2.16 Mups (Million updates per second) to 107.75 Mups with a mean of 18.64 Mups, while the industry standard is only 100 Kups.

The cost of US is an additional step for the lookup. After looking up the FIB compressed by US, suppose the length of the matched prefix is n, we check the $n+1$th bit in the incoming IP address: if it is 0, we report the *leftport*; if it is 1, we report the *rightport*.

US can work perfectly with existing FIB compression algorithms and IP lookup algorithms. Simulation results show that about 7% of prefixes can be reduced when applying US to the optimal compression algorithm ORTC. US alone can not compete with ORTC in terms of compression, but the combination of US and ORTC can beat ORTC in compression (but the combination of US and ORTC can not be updated easily). Around 35% on-chip memory can be saved when applying US to existing well-known IP lookup algorithms. Although some conventional compression algorithms can also be applied to existing IP lookup algorithms, the negative effect is that the update overhead will be aggravated significantly

[1] Trie is a classic data structure to represent a FIB.

[2] Prefix nodes refer to the trie nodes with next hops.



(a) Unite models

(b) Split models. Node A, B, and C are split nodes.

Fig. 2. The models of the US algorithm.

after compression. In contrast, since the worst case of US update is small bounded, the update complexity after US compression stays the same as that before US compression.

1.4. Key novelties

FIB compression is a well studied field, and there have been various solutions in the literature. It seems there is very limited room for further improvements. Conventional FIB compression algorithms compress the prefix nodes with the same next hops, and each prefix node is still related to one next hop after compression. In contrast, we find a new way to compress the number of prefixes by allowing that each prefix node is related to two next hops. Specifically, we change the conventional "one next hop per prefix node" structure into a "two next hops per prefix node" structure. In other words, in our algorithm, the prefix and next hop information of child nodes are united to their parent node.

With regard to update, given an update message, conventional compression algorithms often compress the sub-trie rooted at the updating node, and the worst case is to re-compress the whole trie. In the US algorithm, we constrain the unite operations in two adjacent levels of the trie. Thus, given any update message, the worst case is to update three nodes.

Paper organization: the rest of the paper is organized as follows. Section 2 introduces our proposed US algorithm. Section 3 describes the update and lookup algorithm of US. Section 4 shows the application of US to FIB compression and IP lookup algorithms. Section 5 evaluates the performance of US. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2. Proposed solution

2.1. Background

FIB, trie and nodes: Given an incoming packet, the Forwarding Information Base (FIB) is searched to decide which egress port (*i.e.*, next hop) the packet should be forwarded to. Each FIB entry includes at least two fields: prefix and next hop. Binary trie [25] is a classic data structure to store a FIB. Each FIB entry is represented by a `prefix node` in the trie. The path from the root node to the prefix node corresponds to the prefix, and the corresponding next hop is stored in the prefix node. We call a node without a next hop an `empty node`. We define the `level` of a node as its hop-count distance to the root node whose level is 0. The level of a prefix node is equal to the length of the corresponding prefix. The nodes without child nodes are called *leaf nodes*, while others are called *internal nodes*.

2.2. Rationale

There are mainly three metrics for FIB compression algorithms: compression time, update cost, and memory usage. ORTC achieves the optimal compression ratio in terms of number of prefix nodes at the cost of complicated update and long compression time. As mentioned above, when handling the updates, the lookup process is forced to be suspended, and the incoming packets are buffered. Only if the worst case of update time is small bounded, the risk of packet loss can be eliminated fundamentally. Towards this goal, this paper manages to strike a good trade-off among compression time, update cost, and memory usage.

Conventional FIB compression algorithms are based on the following principle: when two sibling nodes have the same next hop h, they can be represented and replaced by their parent node with next hop h. If all the prefix nodes in the trie have different next hops, conventional algorithms can hardly achieve any compression effect.

In contrast to conventional compression algorithms, our US algorithm strives to make compression even if all the prefix nodes have different next hops. Specifically, given three nodes: q and q's two child nodes (namely q_1 and q_2), we unite q_1 and q_2 into their parent node q with two next hops. In this way, the number of prefixes is reduced to one. However, we do not perform such compression when only one of the three nodes has a next hop, because the unite operation only brings additional update overhead in this case. *The essential difference between US and conventional FIB compression algorithms is that the compression effectiveness of US no longer depends on the next hop similarity*[3]. If all the prefixes have different next hops, compression algorithms such as ORTC can hardly lead to any compression, while US can still compress the prefixes a lot. This is a distinct feature which cannot be supported by any conventional approaches. Note that US can also be combined with various pre-processors to achieve better compression performance (detailed in Section 4.1). In this situation, if

[3] Next hop similarity means that in a small sub-trie, many prefix nodes share the same next hops.

the pre-pocessor depends on next hop similarity, the compression performance will vary in different scenarios (*i.e.*, higher next hop similarity leads to better compression ratio).

2.3. Router architecture

Before going to the details of the US algorithm, we first show how it operates in a real router. Fig. 1 shows the architecture of a router. It consists of the control plane and the data plane. The control plane stores the RIB (Routing Information Base) containing all IP routing information. The prefixes and their selected next hops (a subset of the RIB) constitute the original FIB. Then the original FIB is compressed by US into a compressed FIB which is stored in the control plane. For the sake of fast update, the compressed FIB contains full information (*i.e., oldports, leftports*, and *rightports*). In the data plane, each line card has one copy of the compressed FIB which only contains *leftports* and *rightports*. Given an incoming packet, the line card looks up the FIB, gets a next hop, and then forwards it through the switch network. Each line card has a queue to buffer the incoming data packets.

As shown in Fig. 1, when an update message arrives, first the RIB stored in the control plane will be updated by the routing protocol. If this leads to any FIB update, the update algorithm of US will be applied to the compressed FIB containing full information stored in the control plane. This will result in changes of several prefix trie nodes. These changes are installed in the FIBs (only with necessary information for lookup) stored in the line cards of the data plane. During the installation process, the incoming data packets are buffered in the queue and cannot be forwarded. If there are too many prefix changes to handle per routing update, the FIBs in the line cards cannot be updated fast enough. In this case, the queues may overflow and packet loss may happen. The superiority of US lies in that: it can make the FIB in the line card as small as possible, while ensures fast update speed with controlled worst case update time.

2.4. Unite and split algorithm

The US algorithm traverses the trie two times. First US traverses the trie in postorder and performs unite operations according to four unite models. After the first traversal, there are two kinds of prefix nodes: the nodes with two next hops and the nodes with only one next hop. The nodes with one next hop will be split in the second traversal. We present the details of the four unite models and the split operation as follows.

2.4.1. Unite models

As shown in Fig. 2(a), given a node q, its left child node q_1 and its right child node q_2, there are four unite models. First, q, q_1 and q_2 all have next hops. Second, only q and q_1 have next hops. Third, only q and q_2 have next hops. Fourth, only q_1 and q_2 have next hops. These four models are united into the same result: q_1 and q_2 have no next hops, and q has two next hops: 1|2, where “1” is the next hop of q_1 (or q), and “2” is the next hop of q_2 (or q). After the unite operations, the nodes with two next hops are called `united nodes`, and the nodes that participate in the unite operations other than the united nodes are called *participator nodes*. In Fig. 2(a), the nodes marked with dashed circles are united nodes, and the nodes marked with squares are participator nodes. To control the update time, every trie node can participate in at most one unite operation, and the nodes modified by each unite operation are confined in two adjacent levels.

2.4.2. Split operation

As aforementioned, there are two kinds of prefix nodes after the first traversal: united nodes and nodes with only one next hop. In the second traversal, we conduct split operations on the nodes with only one next hop. The split operation is very simple: just change the next hop h into two next hops $h|h$. The prefix nodes that are split are called *split nodes*. In Fig. 2(b), we show three cases of the split operations. Node A, B, and C are split nodes.

In summary, US performs unite operations in the first traversal, and performs split operations in the second traversal. The pseudo codes of the unite and split operations are shown in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1: Unite(TrieNode * q)

```
if q is NULL then
    return;
$q_1$ = q→left;
$q_2$ = q→right;
Unite($q_1$);
Unite($q_2$);
if two or three nodes of q, $q_1$, $q_2$ have next hops and are not united nodes then
    perform the unite operation according to the four unite models;
```

Algorithm 2: Split(TrieNode * q)

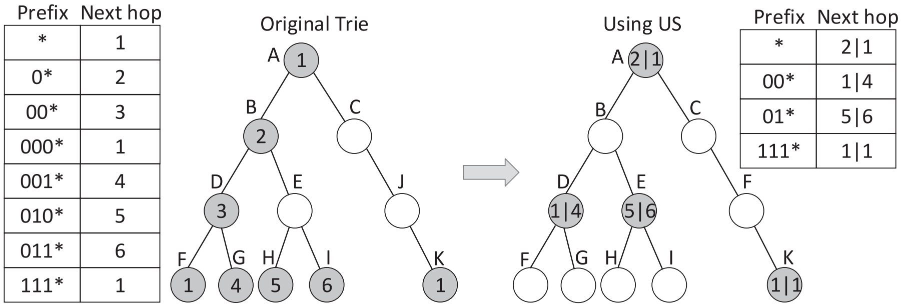
```
if q is NULL then
    return;
Split($q$ →left);
Split($q$ →right);
if q is a prefix node with only one next hop then
    split the next hop of q;
```

2.4.3. Example

We now give an example of the US algorithm in Fig. 3. It shows the original trie and the trie after US compression. Specifically, during the first traversal, node F and G are united into node D with two next hops 1|4; node H and I are united into node E with two next hops 5|6; node A and B are united into node A with two next hops 2|1. After the first traversal, node K with next hop 1 is left behind. During the second traversal, the next hop of node K is split. As a result, node K has two next hops 1|1. In this example, the number of prefixes is reduced from 8 to 4 after US compression.

Note that in this example, the empty leaf nodes (F, G, H, and I) are not deleted in the control plane after compression for the sake of fast update, as their *oldport*s are not empty. The whole trie structure with oldports is kept in the control plane so that we can know exactly what the trie before compression looks like when handling an update message. In the data plane, we only store the prefixes with leftports and rightports, while do not store the empty leaf nodes. In other words, we reduce the usage of fast memory in the data plane at the cost of more usage of slow memory in the control plane.

Actually, our US algorithm can compress the FIBs without tries. One straightforward way is to sort the prefixes, and then do compression for the adjacent prefixes. However, this method will incur complicated update.



Prefix	Next hop
*	1
0*	2
00*	3
000*	1
001*	4
010*	5
011*	6
111*	1

(a) The orignal Trie and FIB

Prefix	Next hop
*	2|1
00*	1|4
01*	5|6
111*	1|1

(b) The Trie and FIB after compressed by US

Fig. 3. An example of US.

3. Update and lookup algorithm of US

3.1. Update algorithm of US

There are two kinds of update messages: announcement and withdrawal. Given an announcement message: [announce *p: h*], it means that we should either insert a new prefix *p* with next hop *h* or change the next hop of the existing prefix *p* to *h*. Given a withdrawal message: [withdraw *p*], it means we should delete the prefix *p* and its corresponding next hop. In practice, the update operations are usually performed in the trie. In the following, we discuss the incremental update algorithm of US for the announcement and withdrawal messages separately.

3.1.1. Announcement handling

To support update, we define three kinds of ports for every trie node: *oldport, leftport, and rightport*, where *oldport* refers to the next hop before compression, *leftport* and *rightport* are the two next hops after compression. The update algorithm for an announcement message [announce *p: h*] proceeds in two steps: first, we set the *oldport* of n_p to *h*, where n_p refers to the corresponding trie node of prefix *p*; second, we update the *leftport* and *rightport* fields according to the node type (united nodes, split nodes, etc.) of n_p. Specifically, there are four cases as follows.

- Case I: n_p *is a united node.* There are three situations: 1) when the two child nodes of n_p are both participator nodes, the *leftport* and *rightport* of n_p keep unchanged; 2) when only the left child of n_p is a participator node, the *rightport* of n_p is changed to *h*; 3) when only the right child of n_p is a participator node, the *leftport* of n_p is changed to *h*.
- Case II: n_p *is a participator node.* Assume the parent node of n_p is $pa(n_p)$. If n_p is the left child of $pa(n_p)$, the *leftport* of $pa(n_p)$ is set to *h*. If n_p is the right child of $pa(n_p)$, the *rightport* of $pa(n_p)$ is set to *h*.
- Case III: n_p *is a split node.* In this case, both the *leftport* and *rightport* of n_p are set to *h*.
- Case IV: n_p *is an empty node or a new node.* First, we need to check whether n_p can be united with its sibling or parent node. If n_p can be united, the *leftport* and *rightport* of $pa(n_p)$ are updated. In this situation, $pa(n_p)$ becomes a united node, and n_p becomes a participator node. Otherwise, n_p should be split, and both the *leftport* and *rightport* of n_p are set to *h*.

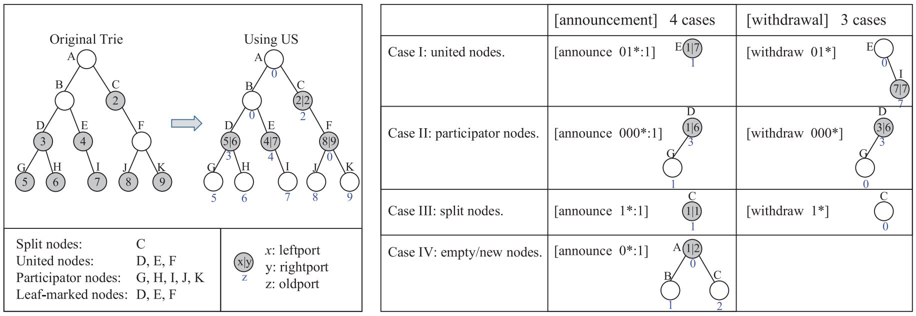
Examples: In Fig. 4, we show four examples which correspond to the above four announcement cases, respectively. Example 1: [announce 01*:1]. It means to change the next hop of united node E to 1. First we change E's *oldport* to 1. Since only the right child of E is a participate node, we just need to set E's *leftport* to 1. Example 2: [announce 000*:1]. It means to change the next hop of participator node G to 1. We set G's *oldport* to 1. Since G is the left child of D, we set D's *leftport* to 1. Example 3: [announce 1*:1]. It means to change the next hop of split node C to 1. In this case, we just set C's *oldport, leftport* and *rightport* to 1. Example 4: [announce 0*:1]. It means to change the next hop of empty node B to 1. In this case, first we set B's *oldport* to 1. Then we find that B's sibling node is a split node, thus B and C can be united: first set the *leftport* and *rightport* of both B and C to 0, then set A's *leftport* to 1, and set A's *rightport* to 2.

3.1.2. Withdrawal handling

Given a withdrawal message: [withdraw *p*], the node n_p corresponding to *p* must be a prefix node and should be deleted from the trie. First, the *oldport* of n_p is set to be empty. Then the *leftport* and *rightport* fields are updated according to the node type of n_p. Specifically, there are three cases as follows.

- Case I: n_p *is a united node.* If both the two child nodes of n_p are participator nodes, the *leftport* and *rightport* of n_p keep unchanged. In this situation, n_p is still a united node. If only one of the child of n_p is a participator node, it can no longer be united to n_p. Thus, the child node needs to be split, and n_p becomes an empty node: the *leftport* and *rightport* of n_p are set to be empty. Note that it is possible that the split child node can be united with its child nodes, this unite operation is not performed in our incremental update algorithm in order to guarantee that at most 3 nodes are changed for any update. In this way, the number of prefixes cannot always stay optimal during the update, but the sacrificed compression ratio is negligible. This conclusion is testified by Fig. 11, in which the size of the compressed FIB almost keeps unchanged during the one-day update.
- Case II: n_p *is a participator node.* There are two situations: 1) the sibling of n_p is also a participator node; 2) the sibling of n_p is not a participator node. In the first situation, if the *oldport* of the parent node $pa(n_p)$ is not empty, the sibling node can be united into $pa(n_p)$; otherwise, the sibling node must be split, and $pa(n_p)$ becomes an empty node. In the second situation, the parent node $pa(n_p)$ must be split. In both situations, n_p becomes an empty node.
- Case III: n_p *is a split node.* In this case, both the *leftport* and *rightport* of n_p are set to be empty, and n_p becomes an empty node.

Example: In Fig. 4, we show three examples which correspond to the above three withdrawal cases, respectively. Example 1: [withdraw 01*]. It means to delete the united node E. Because E has only one child node I which is a participator node, I can no longer be united to E. Therefore, node I should be split. Specifically, we set E's *oldport, leftport*, and *rightport* to 0, and set I's *leftport* and *rightport* to 7. Example 2: [withdraw 000*]. It means to delete the participator node G. Note that for G's parent node D, we



	[announcement] 4 cases	[withdrawal] 3 cases
Case I: united nodes.	[announce 01*:1]	[withdraw 01*]
Case II: participator nodes.	[announce 000*:1]	[withdraw 000*]
Case III: split nodes.	[announce 1*:1]	[withdraw 1*]
Case IV: empty/new nodes.	[announce 0*:1]	

Fig. 4. An example for US update. G and H are united into D, I is united into E, J and K are united into F, C is split. For the sake of fast update, we do not delete G, H, I, J, and K. For the sake of fast lookup, we mark node D, E and F as leaf nodes. For each update message, we do not show the whole trie after update, but only show the changed part in the right table.

Table 1
Comparison on time complexities of update.

Solution	Time complexity
Binary trie without compression	$O(W)$
ORTC	$O(c*(m+n))$
EAR	$O(m+n), O(n)$
4-level	$O(n)$
SMALTA	$O(m+n)$
US	$O(W)$

keep its next hop before compression in the variable *oldport*. Since G's sibling H can still be united into D, we set G's *oldport* to 0, and set D's *leftport* to D's *oldport* 3. `Example 3`: [withdraw 111*]. It means to delete the split node C. In this case, we set C's *oldport, leftport*, and *rightport* to 0.

3.2. Update performance analysis

For US, when updating a node, at most three nodes need to be changed, while other nodes are not affected. Therefore, the worst case of update time is small bounded. It can be concluded that the update time complexity of US is $O(W)$, where W is the maximum depth of the trie. We compare the update complexity of US with several classic compression algorithms in Table 1. In the sub-trie rooted at the updating node, we use n to represent the number of prefix nodes, use m to represent the number of *deleting nodes*[4], and use c to represent the number of distinct next hops. W is 32 for IPv4 FIBs, while n can be pretty large. In the worst case, n is the number of the prefix nodes in the whole trie when the root node is updated. A detailed analysis of update complexity of EAR and ORTC can be found in [16]. The 4-level algorithm needs to rebuild the sub-trie rooted at the updating node, thus its time complexity is O(n). The SMALTA [19] algorithm uses ORTC to take snapshots, thus it has the same compression complexity as ORTC. When inserting or deleting a prefix N, SMALTA restores all the compressed nodes in the sub-trie T_N rooted at node N, so as to correctly perform update. Therefore, many prefix nodes in the sub-trie T_N are decompressed, incurring the trie to be not optimal. Although SMALTA only needs to restore the compressed nodes, it needs to judge whether each node in the sub-trie T_N is compressed or not. Therefore, it often needs to check all the nodes in the sub-trie T_N when updating node N. Furthermore, some prefix nodes in the sub-trie T_N are deleted during compression, thus these `deleting nodes` need to be re-created during update process. Therefore, the update complexity of SMALTA is $O(m+n)$. We conclude that the update performance of US is the same as that of the original binary trie, and significantly outperforms other compression algorithms.

3.3. US lookup algorithm and complexity analysis

The lookup of US abides by the Longest Prefix Matching (LPM) rule [25]. Different from the lookup of the original binary trie, US lookup needs to choose one of the two next hops for each prefix node. Specifically, the lookup of US proceeds in the following steps.

Step I, initialization. Given an incoming IP address s, we define a variable `h` to store the next hop. Initially, we assign the *oldport* of the root node to `h`.

Step II, we obtain the first bit of s, 1) if it is 1, we judge whether the *rightport* of the root node is not empty: if yes, we assign the *rightport* to `h`; otherwise, go to step III. 2) if it is 0, we judge whether the *leftport* of the root node is not empty: if yes, we assign the *leftport* to `h`; otherwise, go to step IV.

Step III, go to the right child node, then obtain the next bit of s, and perform the procedure which is similar to step II. If the current node is a leaf node, the algorithm ends.

Step IV, go to the left child node, then obtain the next bit of s, and perform the similar procedure of step II. If the current node is a leaf node, the algorithm ends.

The pseudo codes of the lookup algorithm of US are shown in Algorithm 3. In the pseudo codes, `root` means the root node of the trie, `IP` means the decimal value of the incoming IP address, and the lookup result is stored in `h`. Obviously, the time complexity

```
Algorithm 3: US_Lookup (root, IP)
p = root;
i = 0;
while $p \neq NULL$ and p.flag is not LEAF do
    if $IP << i >> 31$ then
        if $p \rightarrow rightport > 0$ then
            $h = p \rightarrow$ rightport;
        $p = p \rightarrow$ rchild;
    else
        if $p \rightarrow leftport > 0$ then
            $h = p \rightarrow$ leftport;
        $p = p \rightarrow$ lchild;
    i++;
return h;
```

[4] During the compression process of EAR, ORTC and SMALTA, some leaf nodes are deleted, and they need to be restored during decompression. We call these nodes `deleting nodes`.

of US lookup in the worst case is $O(W)$, which is the same as that of the original binary trie lookup. Many prefix nodes are united to their parent nodes after US compression, which leads to fewer memory accesses during lookup. Thus, the average lookup speed of US is faster than that of the original binary trie. The simulation results are shown in Section 5.4.

4. Applications of US

4.1. Application to existing FIB compression algorithms

In practice, US can be combined with many other FIB compression algorithms to achieve further enhanced compression performances. As analyzed above, the update speed of US is fast and the worst case update time is small bounded. Thus, applying US after other compression algorithms will bring little and fixed extra update overhead. Among ORTC, 4_level[22], EAR [16], and auto aggregation [21], ORTC [18] achieves the best compression ratio. Although NSFIB [24] can achieve a better compression ratio than ORTC, the cost is changing the forwarding behavior. Entropy compression [23] pursues to achieve the lower bound of the information entropy, but the compressed result cannot work with prior IP lookup algorithms. US can be applied to the above FIB compression algorithms. Here we apply US to ORTC for the sake of efficiency and practicality. Given a FIB, we first construct a trie, and then compress it using ORTC and get the resulting trie. We further compress the resulting trie using the US algorithm, and get the final compression result. The related simulation results are shown in Section 5.2.

Here we need to clarify why the compression ratio of ORTC can be further improved. Given the premise that one prefix can only have one next hop and no changes of forwarding behavior happen during compression, ORTC compression is optimal in terms of number of prefix nodes. Given the premise that one prefix can have two next hops, the combination of ORTC and US can achieve a better compression ratio in term of number of prefix nodes. The number of prefixes determines the on-chip memory usage of many IP lookup algorithms, thus we use it as the metric.

4.2. Application to classical IP lookup solutions

The US algorithm can also be used to reduce the on-chip memory usage of IP lookup solutions with little additional update overhead. The on-chip memory is usually small, fast and expensive, thus reducing the on-chip memory usage can significantly reduce the cost and improve the system efficiency. Generally, IP lookup solutions can be divided into two categories: software based solutions and hardware based solutions. Software based solutions include Lulea [14], LC-trie [26], Tree Bitmap [27], and SAIL [11], *etc*. Hardware based solutions include using TCAM (such as coolcam [3], parallel TCAMs [5]), using FPGA (such as [7]), using both TCAM and FPGA (such as hybrid lookup [28]), and using Bloom filters (such as PBF [29], BF for IPv6 lookups [30]). Among these solutions, we apply the US algorithm to three representative ones: PBF [29], hybrid lookup [28], and SAIL [11]. After applying US, the lookup complexity of these fast lookup algorithms keeps unchanged, while the on-chip memory usage is significantly reduced.

Applying US to PBF: for the prefix nodes at each level of the trie, PBF builds one Bloom filter (BF) and one hash table. The BFs are small enough to be stored in the on-chip memory of FPGA [31], while the hash tables are stored in the off-chip memory. Under a certain false positive probability, there is a positive correlation between the size of a BF and the number of prefixes inserted into the BF. By applying US compression, the on-chip memory usage of BFs can be reduced because the number of prefixes is reduced. As for the off-chip hash tables, after US compression, each hash bucket stores one prefix and two next hops. Since the number of hash buckets is reduced, the total size of the off-chip hash tables is reduced. The related simulation results are shown in Section 5.5.

Table 2
On-chip memory usage comparison.

	SAIL_B	SAIL_U	SAIL_L	SAIL_M
before using US	=4MB	$\leq$ 2.03MB	$\leq$ 2.13MB	$\leq$ 2.13MB
after using US	=2MB	$\leq$ 1.016MB	$\leq$ 1.07MB	$\leq$ 1.07MB

Applying US to hybrid lookup: A trie can be partitioned into two parts - the leaf nodes and the trimmed trie. The prefixes of the leaf nodes are stored in TCAM. The trimmed trie is stored in the SRAM-based pipeline of FPGA. After compressing the trie using US, both the number of leaf nodes and the size of the trimmed trie are significantly reduced, thus lead to less memory consumption in both TCAM and FPGA. The related simulation results are shown in Section 5.6.

Applying US to SAIL: SAIL includes four algorithms. SAIL_B is the basic lookup algorithm. For level i of the trie ($0 \leq i \leq 24$), SAIL_B builds a bit map with the length of 2^i, and each prefix node at level i corresponds to a “1” bit in the bit map. SAIL_B stores the bitmaps at level $0 \sim 24$ in the on-chip memory, and thus the upper bound of the on-chip memory usage is $\sum_{i=0}^{24} 2^i = 2^{25} bit = 4$ MByte. Three optimizations based on SAIL_B in terms of update, lookup, and handling multiple FIBs are SAIL_U, SAIL_L, and SAIL_M, respectively. After using US, most prefix nodes at level 24 are united to level 23, thus we just need to store the bitmaps at level $0 \sim 23$ in the on-chip memory, thus the on-chip memory usage is reduced to a half. The worst case of on-chip memory usage of the four algorithms before and after using US is shown in Table 2.

Our algorithm can enhance the cache behavior during IP lookups. For example, assume there are two prefix nodes A and B. As node A and B are often stored separately, the traffic which hits prefix A or B probably does not have good cache behavior. After using our US compression algorithm, node A and B are compressed into one node, and the next hops of A and B are stored adjacently. Therefore, the traffic which hits A or B will have better cache behavior.

4.3. Feasibility analysis

Legacy routers usually use old TCAMs with small capacity. When the FIB size is close to the TCAM capacity, the TCAM needs to be upgraded when using no compression algorithm. Fortunately, our US algorithm can be used to reduce the memory usage of TCAM. After using our US algorithm, the number of prefixes is compressed to about 65% of that of the original FIB. In other words, there will be 35% available memory in the TCAM, and the lifetime of legacy routers can be significantly extended.

The cost of the US algorithm is that during lookup a second step is needed to obtain the exact next hop for a particular packet, since one prefix is shared by two next hops now. When the US algorithm is applied to hardware routers using TCAMs, we can just use a TCAM chip to output the longest matched prefix and a pointer to the corresponding next hop pair, and use FPGA to conduct the extra logic to choose one next hop. Since FPGA is often used in real routers [32], there is no need to add new hardware for packets lookup. Specifically, we store all the next hop pairs in the SRAM of the FPGA. The output pointer of TCAM points to the corresponding next hop pair in FPGA. FPGA chooses the correct next hop by reading an additional bit of the IP address. The extra logic in FPGA only needs one memory access of the SRAM. The lookup speed of TCAM is slower than that of SRAM [29,33,34]. As mentioned in [33], the maximum clock rates of SRAM and TCAM are

Table 3
FIBs used in the simulations.

Router ID	Location	# IPv4 prefixes
rrc00	RIPE NCC, Amsterdam	532,766
rrc01	LINX, London	508,889
rrc03	AMS-IX, Amsterdam	510,680
rrc04	CIXP, Geneva	512,893
rrc05	VIX, Vienna	507,622
rrc10	Milan, Italy	507,894
rrc11	New York, USA	509,738
rrc12	Frankfurt, Germany	518,739
rrc13	Moscow, Russia	552,684
rrc14	Palo Alto, USA	514,408
rrc15	Sao Paulo, Brazil	516,721

400 MHz and 266 MHz, respectively. That's to say, one memory access in SRAM only needs 2.5 ns, while one clock cycle of TCAM needs 3.76 ns. With pipeline, adding the second step will increase the lookup latency. However, as the second step is faster than looking up TCAM, the bottleneck of system throughput lies in the lookup of TCAM. In other words, adding the second step increases the system latency, but do not affect the system throughput.

As TCAMs are expensive and power-hungry, recent significant work, such as SAIL [11], DXR [35], and Poptrie [12] prefer to use software methods to conduct IP lookup. For the software solutions, it is fairly easy and fast to implement the second step: read one additional bit, and then choose one of the two next hops. Section 5.4 shows that the lookup speed of the compressed trie is faster than that of the original trie, since compression leads to shorter lookup path and better cache performance. Therefore, the US algorithm can be easily applied to software routers, and can also be applied to hardware routers.

5. Evaluation

In this section, we first evaluate the compression performance of US and its application to ORTC. Second, we compare the update performance of US with that of the original binary trie. Third, we evaluate the IP lookup speed of the binary trie before and after US compression. Fourth, we show the on-chip memory usage of PBF and hybrid lookup before and after applying the US algorithm. As for SAIL, since there are four SAIL algorithms, we only show the theoretical memory upper bounds in Table 2.

5.1. Simulation setup

FIBs and Updates: We downloaded 11 BGP FIBs from the RIPE RIS Project [36] at 8:00 AM on September 1st 2014. We name these FIBs as $FIBs_{2014}$. We downloaded 12-year FIBs of rrc00 at 8:00[5] AM on September 1st from 2003 to 2014, and name these FIBs as $FIBs_{12years}$. We also downloaded one-day update messages for rrc00, rrc01, and rrc03 starting from 8:00 AM September 1st 2014. Table 3 shows the collecting locations and the sizes of $FIBs_{2014}$.

Synthetic traffic traces: To evaluate the lookup speed of the binary trie before and after US compression, we generate 10M traffic traces based on the simulation FIBs, and guarantee that each prefix is matched with the same probability by the synthetic trace.

Correctness test: We verify the correctness of US by comparing the lookup results of the original FIB and the FIB after US compression using the 4G IPv4 address space. The results are exactly the same, hence our US algorithm passed this correctness test. The appendix contains a proof of the correctness of US using the method proposed in [37].

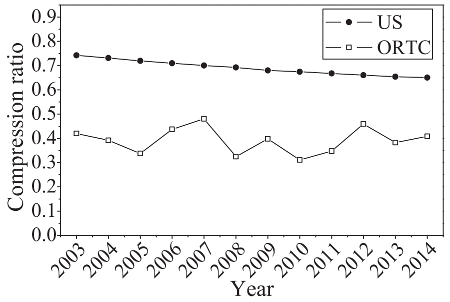


Fig. 5. Compression ratios of US and ORTC over the last 12 years.

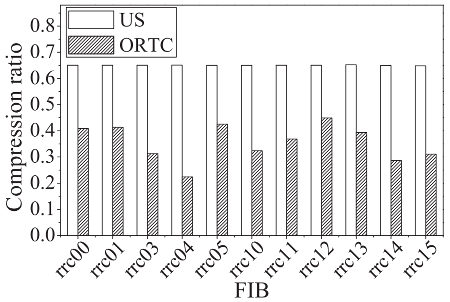


Fig. 6. Compression ratios of US and ORTC for 11 FIBs in 2014.

Computer configuration: We conducted the simulations on a computer with two Intel(R) Core i7-3517U 1.9GHz & 2.4GHz and 8GB RAM running Windows 7 operation system.

5.2. Simulations on FIB compression

We evaluate the compression performance using two metrics: `compression ratio` and `compression time`. Compression ratio is the ratio between the number of prefixes after compression and that before compression. Smaller compression ratio means more reduction of the FIB size. Compression time is the time used to compress the original FIB. Since ORTC achieves the optimal compression ratio, here we evaluate the compression performance of US using ORTC as a baseline.

5.2.1. Compression ratio

Our simulation results show that US compresses the test FIBs by about 35%, and improves the compression ratio of ORTC by about 7% when applied to ORTC. Fig. 5 shows the compression ratio changes of US and ORTC on FIB rrc00 over the last 12 years. As time goes by, the FIB size increases rapidly, and the compression ratio of US gets better steadily. In contrast, ORTC shows an unstable compression ratio. Fig. 6 compares the compression ratio of US and ORTC on $FIBs_{2014}$. US achieves an average compression ratio of 0.65, which means the compressed FIB size is about 65% of the original FIB size. The average compression ratio of ORTC is 0.36. Fig. 7 compares the compression ratio of ORTC and ORTC+US on $FIBs_{2014}$. Results show that combining US and ORTC can reduce the compression ratio of ORTC by about 7%. This does not mean the optimal compression ratio of ORTC is incorrect. *It means that the compression ratio can be further improved when one prefix node stores two next hops*. In sum, although ORTC achieves smaller compression ratios than US, the compression ratio of ORTC+US is smaller than that of ORTC.

[5] There is no available FIB of rrc00 at 8:00 on September 1st 2004 on the website, thus we use the FIB at 16:00 instead.

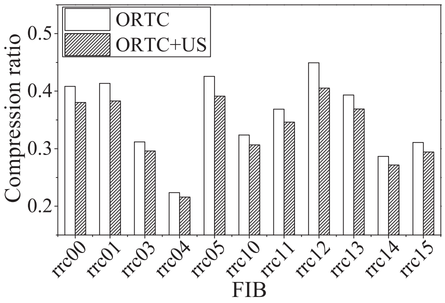


Fig. 7. Compression ratios of ORTC and ORTC+US for 11 FIBs in 2014.

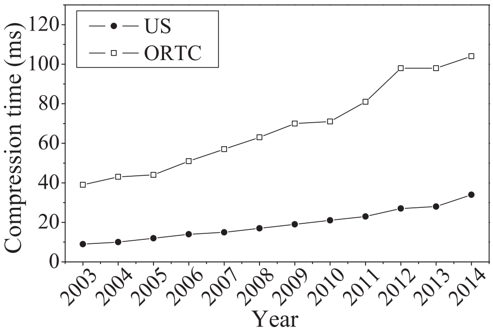


Fig. 8. Compression time of US and ORTC over the last 12 years.

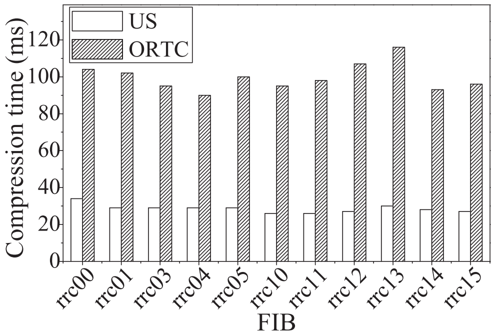


Fig. 9. Compression time of US and ORTC for 11 FIBs in 2014.

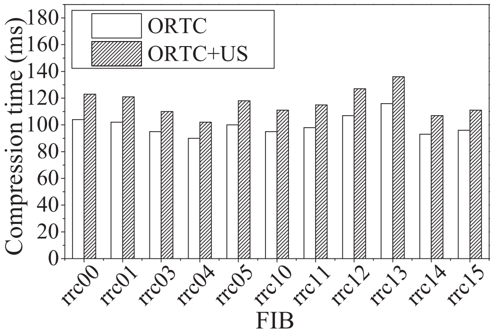


Fig. 10. Compression time of ORTC and ORTC+US for 11 FIBs in 2014.

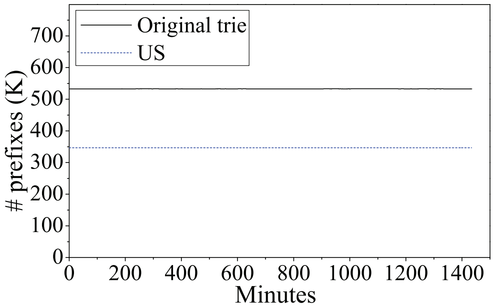


Fig. 11. The growth of prefixes for one-day updates.

5.2.2. Compression time

Our simulation results show that US reduces the compression time by 72% ∼ 77% comparing to ORTC, and the combination of US and ORTC adds little extra compression time overhead. First, as shown in Fig. 8, when using $FIBs_{12years}$, the compression time of US is 23% ∼ 27% of that of ORTC. As time goes by, the compression time increases with the growing size of FIB, and the increase of the compression time of US is much slower than that of ORTC. Second, as shown in Fig. 9, when using $FIBs_{2014}$, the compression time of US is 25% ∼ 28% of that of ORTC. Third, as shown in Fig. 10, when using $FIBs_{2014}$, the compression time of ORTC+US is only 1.13 ∼ 1.16 times of that of ORTC. Note that the compression time of ORTC+US is much less than the sum of ORTC compression time and US compression time. This is because after compressed by ORTC, the trie size is much smaller so that the compression speed of US is faster.

For every trie node, ORTC needs to compute the intersection or union of two next hop sets, and to judge whether a next hop set is a subset of another next hop set. These operations are time-consuming, especially for large FIBs. In contrast, there are only simple assignment and judgment operations in the US compression process. Therefore, the compression speed of US is much faster than that of ORTC.

The US algorithm can be applied to two scenarios. First, for routers with frequent FIB updates, we can use only US algorithm to achieve a good compression ratio with fast update. Second, for some routers with infrequent FIB updates and limited fast memory, we can use ORTC+US to achieve better compression ratio at the cost of slow update.

5.3. Simulations on update

One key advantage of the US algorithm is the fast update with controlled worst case update time. To evaluate the incremental update algorithm of US, we compare the update performance of US with that of the original binary trie, because the update speed of the binary trie without compression is much faster than that of the trie after applying FIB compression algorithms (see Table 1). To evaluate the update overhead of applying US to ORTC, We also compare the update speed of ORTC+US and ORTC in this section. We only show the simulation results for the one-day update of rrc00 which contains 4,906,067 update messages. The simulation results for rrc01 and rrc03 are similar, thus are omitted due to space limitation.

The update algorithm of ORTC+US works as follows. Given a FIB compressed by ORTC+US and an update message, first we locate the updating node. Then the sub-trie *T* rooted at the updating node will be de-compressed and updated. Next, we compress the sub-trie *T* first by ORTC and then by US.

Our simulation results show that the incremental update of US produces 0.08% redundant prefix nodes in one day. Fig. 11 shows how the number of prefixes grows for one-day updates. The x-axes represents the passed minutes in a day. The number of prefixes in the original FIB increases from 532,766 to 532,835 over one-day updates, and the number of prefixes in the compressed FIB increases from 346,464 to 346,757. Both exhibit a very slow increase. Thus we do not need to re-compress the FIB for a very long period of time.

Our simulation results show that the update overhead of the binary trie is lower after US compression. Fig. 12 shows the distribu-

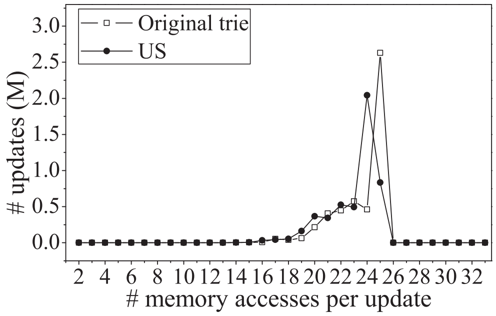


Fig. 12. The distribution of the number of memory accesses per update for one-day updates.

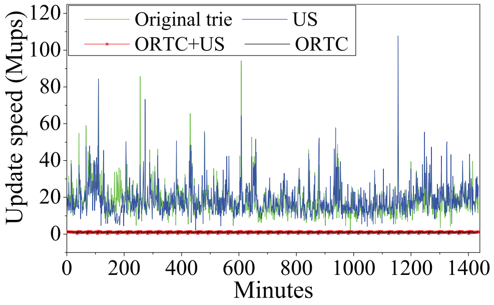


Fig. 13. The update speed for one-day updates.

tion of the number of memory accesses per update for one-day updates. For the original binary trie, results show that 53% updates need 25 memory accesses, because the length of most updating prefixes is 24 and accessing the root node also requires one memory access. After US compression, 44% updates need 24 memory accesses. This is because many prefix nodes at level 24 are united into their parent nodes at level 23.

Our simulation results show that the update speed of US is a little faster than that of the original binary trie, and much faster than that of ORTC. Fig. 13 shows the update speed of the original binary trie, US, ORTC and ORTC+US on one-day update messages. Assume x Mups means x million updates are processed per second. The update speed of the original binary trie ranges from 2.06 Mups to 94.17 Mups with a mean of 18.59 Mups. The update speed of US ranges from 2.16 Mups to 107.75 Mups with a mean of 18.64 Mups. The update speed of ORTC ranges from 0.35 ∼ 1.63 Mups with a mean of 1.08 Mups, while the update speed of ORTC+US ranges from 0.33 Mups to 1.60 Mups with a mean of 1.01 Mups. This indicates that when applying US to ORTC, the update speed remains almost unchanged comparing to that of ORTC. In other words, applying US to ORTC can improve the compression ratio with little additional update overhead.

5.4. Simulations on IP lookup

The worst case lookup complexities of the binary trie before and after US compression are both O(W), where W is the maximum depth of the binary trie. In practice, after using US, the lookup speed is faster because the average number of memory accesses per lookup is reduced after compression. To verify this conclusion, we conduct simulations to evaluate the lookup speed of the binary trie before and after using US.

Our simulation results show that the lookup speed of the binary trie is faster after US compression. First, we lookup $FIB_{12years}$ using the corresponding synthetic traffic traces, and the results are shown in Fig. 14. The lookup speed after using US ranges from 13.9 to 14.8 Mpps (Million packets per second), which is 1.17 ∼ 1.22 times faster than the lookup speed of the original binary trie. Second, we conduct similar simulations on FIB_{2014}, and the results are shown in Fig. 15 . It shows that the lookup speed after US compression is 1.18 ∼ 1.21 times faster than that of the original binary trie.

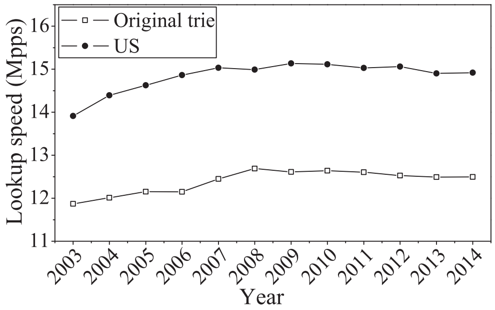


Fig. 14. The lookup speed for rrc00 over the last 12 years.

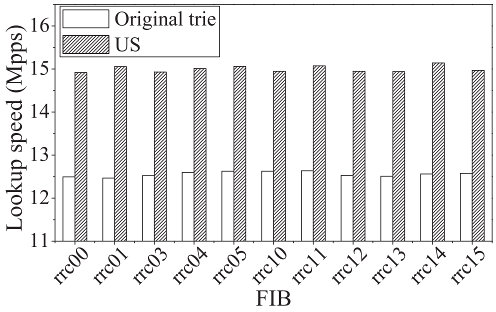


Fig. 15. The lookup speed for 11 FIBs in 2014.

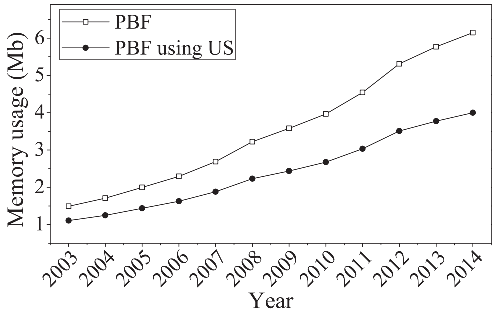


Fig. 16. On-chip memory usage of rrc00 over the last 12 years.

5.5. Simulations on PBF

In this section, we evaluate the memory usage of PBF before and after using US. We set the number of hash functions of all the Bloom filters to 8, and the sum of the sizes of all the Bloom filters is the on-chip memory usage.

5.5.1. On-chip memory usage

Our simulation results show that the on-chip memory usage of PBF is reduced by about 35% after US compression. The on-chip memory usage for $FIBs_{12years}$ is shown in Fig. 16. The on-chip memory usage grows year by year because the number of prefixes increases. The on-chip memory usage of PBF before US compression ranges from 1.5 Mb to 6.2 Mb. When using US, the on-chip memory usage of PBF is reduced to 1.1 ∼ 4.0 Mb, which is 64.5% ∼ 73.3% of the on-chip memory usage before using US. We conduct similar simulations using $FIBs_{2014}$, and the results are shown in Fig. 17. The on-chip memory usage of PBF after using US ranges from 3.81

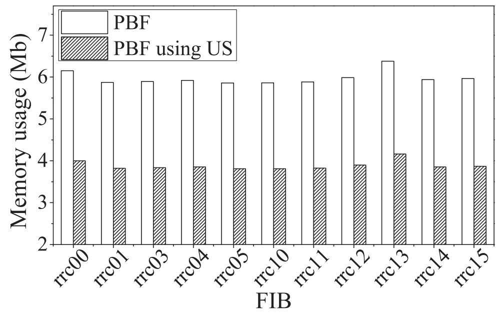


Fig. 17. On-chip memory usage for 11 FIBs in 2014.

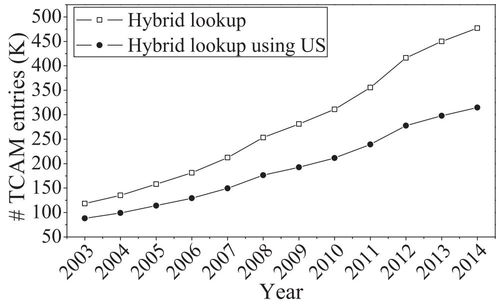


Fig. 18. # prefixes stored in TCAM for rrc00 over the last 12 years.

Mb to 4.16 Mb, and is 64.8% ∼ 65.2% of the on-chip memory usage before using US.

5.5.2. Off-chip memory usage

Our simulation results show that the off-chip memory usage of PBF is reduced by 21.8% ∼ 22.2% *after US compression.* PBF uses hash tables in the off-chip memory. The size of a hash table is proportional to the number of prefixes stored in it. After using US, each hash bucket stores one prefix and two next hops, and the total number of prefixes is reduced. We compare the total size of hash tables before and after US compression using $FIBs_{2014}$.

5.6. Simulations on hybrid lookup

In this section, we evaluate the memory usage of hybrid lookup before and after using US. The update of hybrid lookup includes two parts: updating the trie structure, and updating TCAM or FPGA. The main advantage of hybrid lookup is the $O(1)$ update complexity when the update of the trie is not considered, because updating the trie do not interrupt lookup. The update complexity is still $O(1)$ when applying US to hybrid lookup, because at most three nodes need to be changed for any update of US. The cost and power consumption of the hybrid lookup algorithm increases linearly with the growth of the number of prefixes stored in FGPA and TCAM. Therefore, we can use US to reduce the number of prefixes while keeping the $O(1)$ update performance.

5.6.1. TCAM usage

Our simulation results show that the number of TCAM entries is reduced by about 35% after US compression. We evaluate the TCAM usage (the number of prefixes stored in TCAM), which is also the number of leaf nodes in a trie. Fig. 18 shows the simulation results on $FIBs_{12years}$. It can be observed that the number of TCAM entries is reduced to 65% ∼ 74% after using US. We conduct similar simulations on FIB_{2014}, and the results are shown in Fig. 19. The number of TCAM entries is reduced to 65% ∼ 66% after using US. Note that each prefix (*i.e.*, TCAM entry) corresponds to two next hops, and the next hops are stored in the associated SRAM. Simulation results show that the SRAM usage is 0.45 ∼ 0.49 MB before US compression, and becomes 0.64 ∼ 0.66 MB after US compression. Such small overhead is negligible.

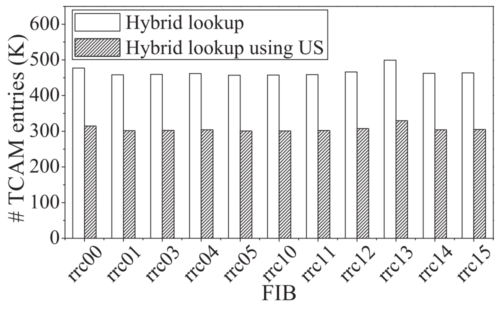


Fig. 19. # prefixes stored in TCAM for 11 FIBs in 2014.

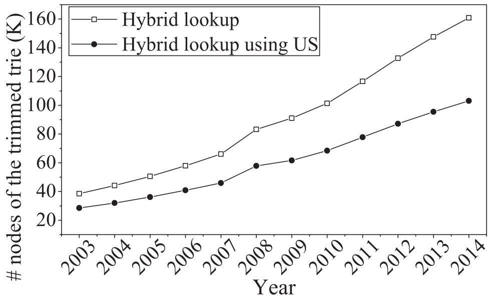


Fig. 20. # trimmed trie nodes stored in FPGA for rrc00 over the last 12 years.

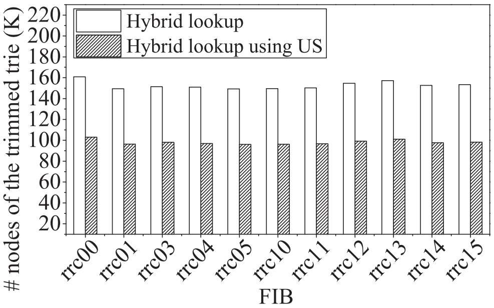


Fig. 21. # trimmed trie nodes stored in FPGA for 11 FIBs in 2014.

5.6.2. FPGA usage

Our simulation results show that the memory usage of FPGA (FPGA usage) is reduced to 71.0% ∼ 72.0% *after US compression.* We evaluate the number of the trimmed trie nodes stored in FPGA. When using $FIBs_{12years}$, as shown in Fig. 20, the number of trimmed trie nodes is reduced to 64% ∼ 74% after using US. When using $FIBs_{2014}$, as shown in Fig. 21, the number of trimmed trie nodes is reduced to 63.9% ∼ 64.8%. We also evaluate the on-chip memory usage of FPGA for $FIBs_{2014}$. Simulation results show that the memory usage of the trimmed trie after US compression is reduced to 71.0% ∼ 72.0%.

6. Related work

FIB compression is a well studied and important issue due to the significance of FIBs in router design. In this paper, we classify FIB compression algorithms into four categories.

The first category compresses a FIB into a smaller one and does not change the forwarding behavior, such as auto aggregation [21], ORTC [18], EAR [16], and wild-card compression [38]. The auto aggregation algorithm [21] only compresses the sibling prefix nodes with the same next hop. This compression algorithm is simple and fast, but its compression ratio is not attractive and the update time is not small bounded. ORTC [18] achieves the theoretical optimal compression ratio in terms of the number of prefixes. It traverses the trie three passes to complete the compression. Generally speaking, the update of one prefix can be implemented by re-constructing the sub-trie rooted at the updating trie node using the same compression algorithm. However, optimal compression is complicated and slow, thus leads to complicated and slow update. The authors in [18] did not present the update algorithm of ORTC. To address the update problem, Liu et al. proposed two incremental update algorithms in [20] for ORTC, and also proposed FIFA-S, FIFA-A and FIFA-H in [17] to further improve the update performance. Uzmi et al. [19] also proposed an update algorithm for ORTC. The main idea is to only update the nodes which are affected, so as to accelerate the update speed. The key reason of slow update is due to optimal compression. Thus sub-optimal compression with fast update becomes an appealing alternative choice. Yang et al. proposed two sub-optimal compression algorithms, EAR_slow and EAR_fast [16]. Compared with ORTC, EAR_fast can reduce the compression time to around 1/10 at the cost of a little compression ratio loss. Although EAR_fast can achieve fast update speed, the worst case update time is still not small bounded. LFA [39] and BLOCK [40] optimize the update performance by leveraging the temporal and spatial locality of udpates. The basic idea of these two algorithms is similar: only the sub-tries which are not updated for a preset period of time are compressed. If an update occurs in a compressed sub-trie, the compressed sub-trie will be forced to be de-compressed/split and then be updated. For the update complexity, the best case of these two algorithms is $O(W)$, and the worst case is $O(n)$, where W is the maximum depth of the trie, and n is the number of nodes in the sub-trie. Different from above compression methods based on binary trie, wild-card compression [38] compresses the prefixes into new ones, and each bit has three states: 0, 1, and don't care. Thus it can only be applied to TCAM-based IP lookup solutions. What is more, its update is really complicated and slow. In summary, the worst case update of all the above FIB compression and update algorithms is to update the whole sub-trie rooted at the updating node. US also belongs to this category, but only needs to change at most three nodes in the worst case of update. The update complexity of US is always $O(W)$.

The second category aims at achieving a better compression ratio at the cost of changing the forwarding behavior, such as NSFIB [24] and 4-level [22]. NSFIB can achieve a much better compression ratio than ORTC by taking advantage of multiple next hops. The 4-level algorithm defines four levels of FIB compression. The first two levels are simple, but the compression is not sufficient. The last two levels achieve better compression ratios at the cost of forwarding some packets which should have been dropped.

The third category focuses on compressing the FIB towards the information entropy bounds, and the compressed result is represented by bits rather than prefixes. Rétvári et al. [23] introduced the information entropy of tries for the first time, and there are two successors. Rottenstreich et al. [41] proposed an encoding scheme to achieve sub-optimal memory requirement, and Korosi et al. [42] focused on improving the lookup speed. The common disadvantage of this category is the complicated and slow update performance.

The above three categories are purely local solutions, and do not affect neighboring routers. The fourth category requires the coordination between routers or between switches and controllers. In [43], three route aggregation strategies are proposed to compress the FIBs. These strategies either require coordination between the ASes or need to change the way routers build their FIBs. A recent work [44] focuses on minimizing the number of updates sent from the controller to the compressed FIBs stored in switches.

7. Conclusion

With the rapid growth of FIB size in backbone routers, FIB compression becomes a hot topic in recent years, and various FIB compression algorithms have been proposed. The update performance will inevitably be degraded if a FIB is compressed. Only when the worst case of update time is small bounded, the risk of packet loss during updates can ultimately be avoided. Towards this goal, we propose the Unite and Split (US) compression algorithm in this paper to achieve fast update with small bounded worst case (*i.e.*, at most three nodes need to be changed per update). Further, we use the US algorithm to improve the performance of several classic software and hardware lookup algorithms. Simulation results on real-world FIBs show that the compression ratio of US is about 65% with fast compression time (only about 28.5 ms), and the update speed of US is fast. In addition, the on-chip memory usage of several classic lookup algorithms is significantly reduced after applying US. To enable others to replicate the simulations, we released the source code of our US algorithm and related data set at Github [45].

Acknowledgements

The research is supported by the National Basic Research Program of China (973 Program) under Grant 2012CB315806, the National Natural Science Foundation of China under Grant 61133015, and Specialized Research Fund for the Doctoral Program of Higher Education under Grant 20120002110060.

Appendix

The authors of [37] proposed a universal method to prove the correctness of FIB compression algorithms. We use this method to prove the correctness of the US algorithm below. First, as mentioned in [37], the prefixes are represented by regular expression syntax, and the symbols used are defined as follows.

- q is a node in the trie, and (q) represents the corresponding prefix of node q. Prefix nodes have next-hops, while empty nodes don't.
- $(q_1 q_2)$ represents the bit string of the path between prefix nodes q_1 and q_2, where no prefix nodes appear in the path.
- $L(q)$ represents the prefix length of node q.
- P represents a trie and (q) represents a prefix, then $P(q)$ means the next-hop of prefix (q) in trie P.
- $(\overline{q})$ represents a prefix with the same length of (q), but it is different from (q). $P(\overline{q})$ means the next hop of prefix $(\overline{q})$ in trie P.
- $(a|b)$ represents a prefix with two next hops. Its leftport is a, and rightport is b. Given one more bit 0 or 1, there is $(a|b)_0 = a$ and $(a|b)_1 = b$.

The operation of XOR is defined as follows:

$\forall x, y \in G$

$$x \oplus y = \begin{cases} x + y, & xy(x+y) = 0 \\ y, & x > 0, y > 0 \\ meaningless, & otherwise \end{cases}$$

∀ IP address R, R=[0,1]{32}, suppose the match result of the most significant i bits is S_i, then the next-hop of R is $P(R) = S_1 \oplus S_2 \oplus \ldots S_{32} = \oplus_{i=1}^{32} S_i$.

Due to space limitation, we only show the proof of the first model (see Fig. A.22) of our US algorithm. The proof of other models is similar.

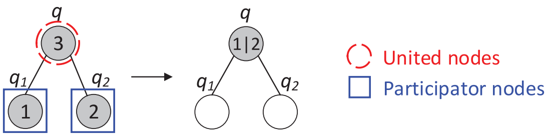


Fig. A.22. The first model of US.

Proof. ∀ IP address R, obviously, $L(R) = K, R = [0,1]K$. Suppose $R = [0,1]L(q)[0,1][0,1]\{K - L(q) - 1\}$.

Step1: matching $[0,1]\{L(q)\}$

$$\left.\begin{aligned}[0,1]\{L(q)\} = (q) \Rightarrow & \begin{cases} P1([0,1]\{L(q)\}) = 3 \\ P2([0,1]\{L(q)\}) = (1|2) \end{cases} \\ [0,1]\{L(q)\} \neq (q) \Rightarrow & \; P1 = P2 \\ & \; P1(\overline{q}) = P2(\overline{q}) \end{aligned}\right\} \Rightarrow \mathit{Only\ considering}\,[0,1]\{L(q)\} = (q)$$

Step2: when [0,1]{L(q)}=(q), matching [0, 1]

$$\left.\begin{aligned}[0,1] = 0, or1 \Rightarrow & \begin{cases} P1(0) = P1(q_1) = 3 \oplus 1 = 1 \\ P1(1) = P1(q_2) = 3 \oplus 2 = 2 \end{cases} \\ & P2(0) = (1|2)_0 = 1 \\ & P2(1) = (1|2)_1 = 2 \end{aligned}\right\} \Rightarrow P1([0,1]) = P2([0,1])$$

At this stage,

$P1([0,1]\{L(q)\})[0,1]) = P1([0,1]\{L(q)\})) \oplus P1([0,1]) = P1([0,1])$

$P2([0,1]\{L(q)\})[0,1]) = P2([0,1]\{L(q)\})) \oplus P2([0,1]) = P2([0,1])$

Thus, $P1([0,1]\{L(q)\})[0,1]) = P2([0,1]\{L(q)\})[0,1])$.

For P1 and P2, they have the same rest parts. In other words, when matching $[0,1]\{K - L(q) - 1\}$, P1 and P2 will report the same results.

According to step1 and step2, $P1 \Leftrightarrow P2$. □

References

[1] BGP routing table analysis reports, (http://bgp.potaroo.net/).
[2] 512k bug[on line], (Available: http://www.theinquirer.net/inquirer/news/2360306/512k-routing-bug-might-kill-the-internet-but-probably-wont).
[3] F. Zane, G. Narlikar, A. Basu, Coolcams: Power-efficient TCAMs for forwarding engines, in: Proc. IEEE INFOCOM, vol. 1, IEEE, 2003, pp. 42–52.
[4] T. Yang, R. Duan, J. Lu, S. Zhang, H. Dai, B. Liu, Clue: achieving fast update over compressed table for parallel lookup with reduced dynamic redundancy, in: Proc. IEEE ICDCS, 2012, pp. 678–687.
[5] K. Zheng, C. Hu, H. Lu, B. Liu, A TCAM-based distributed parallel IP lookup scheme and performance analysis, Netw. IEEE/ACM Trans. 14 (4) (2006) 863–875.
[6] V. Ravikumar, R.N. Mahapatra, L.N. Bhuyan, Easecam: An energy and storage efficient tcam-based router architecture for ip lookup, Comput. IEEE Trans. 54 (5) (2005) 521–533.
[7] H. Le, W. Jiang, V.K. Prasanna, A SRAM-based architecture for trie-based IP lookup using FPGA, in: Proc. IEEE FCCM, IEEE, 2008, pp. 33–42.
[8] H. Le, W. Jiang, V.K. Prasanna, A sram-based architecture for trie-based ip lookup using fpga, in: Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on, IEEE, 2008, pp. 33–42.
[9] H. Le, V.K. Prasanna, Scalable high throughput and power efficient ip-lookup on fpga, in: Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on, IEEE, 2009, pp. 167–174.
[10] W. Jiang, V.K. Prasanna, A memory-balanced linear pipeline architecture for trie-based ip lookup, in: High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on, IEEE, 2007, pp. 83–90.
[11] T. Yang, G. Xie, Y. Li, Q. Fu, A.X. Liu, Q. Li, L. Mathy, Guarantee IP lookup performance with FIB explosion, in: Proc. ACM SIGCOMM, 2014, pp. 39–50.
[12] H. Asai, Y. Ohara, Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup, in: Proc. ACM SIGCOMM, 2015.
[13] S. Nilsson, G. Karlsson, Ip-address lookup using lc-tries, Selected Areas Commun. IEEE J. 17 (6) (1999) 1083–1092.
[14] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, vol. 27, ACM, 1997.
[15] T. Yang, Z. Mi, R. Duan, X. Guo, J. Lu, S. Zhang, X. Sun, B. Liu, An ultra-fast universal incremental update algorithm for trie-based routing lookup, in: Proc. IEEE/ACM ICNP, 2012.
[16] T. Yang, B. Yuan, S. Zhang, T. Zhang, R. Duan, Y. Wang, B. Liu, Approaching optimal compression with fast update for large scale routing tables, in: Proc. IEEE IWQoS, 2012, p. 32.
[17] Y. Liu, B. Zhang, L. Wang, FIFA: Fast incremental FIB aggregation, in: Proc. IEEE INFOCOM, 2013.
[18] R.P. Draves, C. King, S. Venkatachary, B.D. Zill, Constructing optimal IP routing tables, in: Proc. IEEE INFOCOM, 1999.
[19] Z.A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, P. Francis, SMALTA: practical and near-optimal FIB aggregation, in: Proc. ACM CoNEXT, ACM, 2011, p. 29.
[20] Y. Liu, X. Zhao, K. Nam, L. Wang, B. Zhang, Incremental forwarding table aggregation, in: Proc. IEEE GLOBECOM, 2010.
[21] B. Cain, Auto aggregation method for IP prefix/length pairs, 2002, US Patent 6,401,130.
[22] X. Zhao, Y. Liu, L. Wang, B. Zhang, On the aggregatability of router forwarding tables, in: Proc. IEEE INFOCOM, 2010.
[23] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, Z. Heszberger, Compressing IP forwarding tables: Towards entropy bounds and beyond, in: Proc. ACM SIGCOMM, 2013.
[24] Q. Li, D. Wang, M. Xu, J. Yang, On the scalability of router forwarding tables: Nexthop-selectable FIB aggregation, in: Proc. IEEE INFOCOM, IEEE, 2011, pp. 321–325.
[25] M.A. Ruiz-Sanchez, E.W. Biersack, W. Dabbous, Survey and taxonomy of IP address lookup algorithms, Netw. IEEE 15 (2) (2001) 8–23.
[26] S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, Selected Areas Commun. IEEE J. 17 (6) (1999) 1083–1092.
[27] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, ACM SIGCOMM Comput. Commun. Rev. 34 (2) (2004) 97–122.
[28] L. Luo, G. Xie, Y. Xie, L. Mathy, K. Salamatian, A hybrid IP lookup architecture with fast updates, in: Proc. IEEE INFOCOM, 2012.
[29] S. Dharmapurikar, P. Krishnamurthy, D.E. Taylor, Longest prefix matching using bloom filters, in: Proc. ACM SIGCOMM, 2003.
[30] H. Song, F. Hao, M. Kodialam, T. Lakshman, IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards, in: Proc. IEEE INFOCOM, IEEE, 2009, pp. 2518–2526.
[31] FPGA data sheet[on line], (Available: http://www.xilinx.com).
[32] Cisco website, (http://www.cisco.com/c/en/us/td/docs/routers/asr1000/install/guide/asr1routers/asr1higV8/asr1_hw2.html).
[33] H. Le, W. Jiang, V.K. Prasanna, Scalable high-throughput sram-based architecture for ip-lookup using fpga, in: Field Programmable Logic and Applications, IEEE, 2008, pp. 137–142.
[34] D. Kreutz, F.M.V. Ramos, P. Verissimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: a comprehensive survey, Proc. IEEE 103 (1) (2015) 14–76.
[35] M. Zec, L. Rizzo, M. Mikuc, Dxr: towards a billion routing lookups per second in software, ACM SIGCOMM Comput. Commun. Rev. 42 (5) (2012) 29–36.
[36] RIPE network coordination centre[on line], (Available: http://www.ripe.net/data-tools/stats/ris/ris-raw-data).
[37] T. Yang, G. Xie, et al., A fresh look at forwarding information base compression via mathematical analysis, in: Network Operations and Management Symposium (NOMS), IEEE, 2014, pp. 1–4.
[38] H. Yu, A memory-and time-efficient on-chip TCAM minimizer for IP lookup, in: Proc. IEEE DATE, 2010.
[39] N. Sarrar, R. Wuttke, S. Schmid, M. Bienkowski, S. Uhlig, Leveraging locality for fib aggregation, in: Global Communications Conference (GLOBECOM), 2014 IEEE, IEEE, 2014, pp. 1930–1935.
[40] M. Bienkowski, S. Schmid, Competitive fib aggregation for independent prefixes: Online ski rental on the trie, in: Structural Information and Communication Complexity, Springer, 2013, pp. 92–103.
[41] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, A. Hassidim, Compressing forwarding tables, in: Proc. IEEE INFOCOM, IEEE, 2013, pp. 1231–1239.
[42] A. Korosi, J. Tapolcai, B. Mihálka, G. Mészáros, G. Rétvári, Compressing IP forwarding tables: Realizing information-theoretical space bounds and fast lookups simultaneously, IEEE ICNP, 2014.
[43] J.L. Sobrinho, F. Le, A fresh look at inter-domain route aggregation, in: Proc. IEEE INFOCOM, IEEE, 2012, pp. 2556–2560.
[44] M. Bienkowski, N. Sarrar, S. Schmid, S. Uhlig, Competitive FIB aggregation without update churn, in: Proc. IEEE ICDCS, 2014.
[45] Source code of US[on line], (Available: https://github.com/ussource).

Yuanyuan Zhang is a Ph.D. candidate in the Department of Computer Science and Technology, Tsinghua University. She received the B.Eng. degree from Beijing University of Posts and Telecommunications, China. Her research interests include computer network architecture, network virtualization, and high-speed IP address lookup.

Mingwei Xu is a professor in the Department of Computer Science and Technology, Tsinghua University. He received the B.Eng. degree and the Ph.D. degree from Tsinghua University, China. His research interests include computer network architecture, Internet routing, network virtualization and high-speed router architecture.

Ning Wang is a reader (associate professor) at the Institute for Communication Systems (ICS), University of Surrey. He received his B.Eng (Honours) degree from Changchun University of Science and Technology, P.R. China, in 1996, his M.Eng. degree from Nanyang University, Singapore, in 2000, and his Ph.D. degree from the University of Surrey in 2004. His research interests mainly include future network design, network management and control, and Internet of things (IoT) applications.

Jun Li is an associate professor in the Department of Computer and Information Science, University of Oregon. He received his Ph.D. degree in computer science from the University of California, Los Angeles, CA, in 2002, with honors (Outstanding Doctor of Philosophy). Prior to that, he received his M.E. degree from the Chinese Academy of Sciences in 1995, and his B.S. degree from the Peking University in 1992. His research interests mainly include networking, distributed systems, and their security.

Penghan Chen received the B.Eng. degree from Beijing University of Posts and Telecommunications, China. He is now a Master candidate in the Broadband Network Research Center, Beijing University of Posts and Telecommunications. His research interests include network virtualization and high-speed IP address lookup.

Fei Liang received the B.Eng. degree from Harbin University of Science and Technology, China. He is now a Master candidate in the Broadband Network Research Center, Beijing University of Posts and Telecommunications. His research interests include network virtualization and high-speed IP address lookup.