Longest Prefix Matching with Pruning

Lingtong Liu*, Jun Hu*, Yibo Yan†, Siang Gao†, Tong Yang†, Xiaoming Li†.
* School of Computer Science and Technology, Xidian University, China
† Department of Computer Secience, Peking University, China

***Abstract*—Nowadays, an explosive increase of the size of Forwarding Information Base (FIB) in backbone router makes IP lookup a challenging issue. An effective solution is to use Bloom filter. Because of its false positive, to boost longest prefix matching (LPM), sophisticated Bloom filter-based algorithms were proposed, which makes FIB updating extremely difficult. To address this issue, in this paper, we propose a novel IP lookup algorithm which prunes unneeded prefix probes incurred by false positive and supports FIB updating. Experimental results show that our algorithm can save approximately $7 \sim 24\%$ on-chip memory space than the well-known PBF (longest Prefix matching using Bloom Filters) while keeping the same average number of probes of 1.003 per IP lookup. In other words, given a limited on-chip memory budget, our proposed algorithm can handle FIB $7 \sim 24\%$ bigger than PBF. The source code can be found in [1].**

***Index Terms*—FIB, IP lookup, Bloom filter, pivot pushing**

I. Introduction

With the popularity of the Internet, the size of FIBs on backbone routers takes on an explosive growth, and has almost met $700,000$ [2], making IP lookup a challenging issue. To handle the problems caused by fast growth of FIB size, an elegant IP lookup algorithm should achieve two goals at the same time. On the one hand, IP lookup time, particularly per packet cost, should catch up with the wire speed and remain constant as FIB size grows. On the other hand, on-chip memory usage should meet capacity constraints for the current large FIBs. Compared with off-chip DRAM, access speed of on-chip memory [3] (*i.e.*, CPU cache and FPGA block RAM) is much faster, but its capacity is much smaller and it is much more expensive than DRAM. An IP lookup algorithm not satisfying both of these two goals could hardly be deployed in real routers by ISPs.

Benefiting small memory usage of Bloom filter, Dharmapurikar *et al.* [4] proposed PBF algorithm, which, for the first time, use on-chip Bloom filters to determine the longest matching prefix (LMP, for short). Each Bloom filter encodes sets of prefixes (from FIB records) of the same length and is fitted into on-chip memory for fast membership query. Given an incoming destination address a, not only the length of LMP of a is matched, but also, due to false positive, higher lengths may be returned by Bloom filters. Then, probing of *candidate prefixes* is conducted in descending order of lengths from hash tables (which stores prefixes of the same length in off-chip memory) until a hit happened, returning the next hop of founded LMP of a. Lim *et al.* [5] used one on-chip bloom filter to achieve high-speed IP address lookup. Using precomputation [6] or leaf-pushed trie, Mun *et al.* [7] generated one on-chip bloom filter to find LMP in ascending order of length. Greatly reduced number of probes boosts IP lookup, but the sophisticated trie modification which is used to program Bloom filters makes FIB updating extremely difficult. In 2014, Yang *et al.* [8] proposed pivot pushing algorithm to split prefixes of *an expanded trie* at a selected level which is called *pivot level*, guaranteeing that a found internal node at pivot level indicates the length of the LMP of a no less than pivot level.

Inspired by the pivot pushing, we manage to reduce off-chip memory consumption and update complexity by using a new trie expansion method in initializing Bloom filters, and propose an efficient IP lookup algorithm, longest prefix matching with pruning (LPMP for short), reducing prefix probes and a fast update algorithm. In a nutshell, we carry out revised pivot pushing operation on the original trie constructed using FIB to form an expanded trie which is used to initialize Bloom filters for each prefix length. We also build, at the pivot level, an additional Bloom filter for *hollow nodes* (internal nodes having no next-hop, while *solid nodes* is any node that has the next-hop as its tag) to reduce the number of candidate prefixes. By carefully choosing the appropriate pivot level, our algorithm can performs much better than PBF.

In summary, we make the following major contributions:

1) A revised pivot pushing method is proposed leading us to fast IP lookup and update.
2) An efficient IP lookup algorithm (LPMP) and a fast update algorithm are presented.
3) Extensive experiments are conducted to evaluate the performance of LPMP with PBF.

The rest of this paper is organized as follows. Section II reviews related works. Section III details revised pivot pushing method, the IP lookup and update algorithms separately. Experimental results on our LPMP and PBF are analyzed in Section IV. Finally, we conclude the paper in Section V.

II. Related Work

As a major bottleneck in packet forwarding of today's routers, IP address lookup and update are extensively studied in recent years' papers for better efficiency. Algorithms using a trie to store FIB and handling IP lookup have some variations: binary trie [9], path-compressed trie [10], k-stride multibit trie [11], and full expansion/compression [12], *etc.* Linearly search complexity on address length makes them inefficient. Organizing prefixes into different tables based on their lengths, binary search on lengths [6] were proposed using prefix expansion. As prefix represents a contiguous address space, disjoint ranges of addresses can be sorted for binary range search [9], and multiway range tree [13], reducing test times

978-1-7281-1686-0/19/$31.00 ©2019 IEEE

to some extend, while difficult for fast updating. Hardware features such as parallel processing and memory hierarchy can also be exploited to design IP lookup algorithms. Ternary Content Addressable Memory (TCAM) can be used to query all prefixes in parallel and is a good choice when designing high-speed commercial routers. Due to very limited size and high power consumption, some works [14], [15] managed to reduce the power consumption while achieving faster lookup speed. FPGA-based IP lookup algorithms need to address two main problems: storing the whole FIB in on-chip memory and constructing pipeline stages. Fadishei *et al.* [16] proposed to store part of data structures to improve lookup speed. Exchanging part of the prefixes or rotating some branches [17], [18] adjusted the trie structure to balance stage sizes. With the help of parallel processing capability of GPU, some methods have been proposed to use GPU to improve IP lookup [19], [20]. Fitting frequently searched prefixes of lengths less then a threshold (mostly 24) in fast on-chip memory, [8], [21] used two-level multibit trie to gain a constant off-chip memory IP lookup speed. Using Bloom filters to test the existence of all prefixes in on-chip memory, [4], [5], [7], [22] achieved approximately one off-chip memory lookup speed.

III. Longest Prefix Matching with Pruning

In this section, the revised pivot pushing method, LPMP and fast update algorithms are proposed. Before that, we introduce the layout of core data structures used for IP lookup and update. Symbols and terms used in the following paragraphs are listed in Table I.

TABLE I
Acronyms and symbols used in this paper.

Symbol	Description
W	the number of bits constructing an IP address
a	a binary string denoting an IP address
a_i	the length-i prefix of a
pf	a binary string denoting a prefix
$pf[i]$	the i-th bit of pf
level i	set of length-i prefixes in FIB, or the i-th level of a trie
BM_i	a 2^i-long bitmap used to filter prefixes at level i
BF_i	a Bloom filter used to filter prefixes at level i
BFC_i	a counting Bloom filter assisting update at level i
b	using $b+1$ bitmaps to filter prefixes at level $0 \sim b$
n_i	the number of length-i prefixes in FIB
m_i	the size of BF_i
p	choosing level p as pivot level
BFA_p	a Bloom filter used for pruning at pivot level
$BFCA_p$	a counting Bloom filter assisting update at pivot level

A. Layout of the Data Structures

As we know, to support packet forwarding, a router has two planes: control plane hosting RIB (Routing Information Base) generated during a routing protocol, and data plane hosting FIB. In our setting, as expanded trie can be harnessed guiding fast FIB update, we host it near the RIB in the control plane and generate incremental update information once a new updating requirement arrives. The immediate update information is flowed from control plane to data plane helping FIB updating. Taking advantage of the on-chip off-chip memory hierarchy, we organize the core data structures hosted in the data plane as shown in Figure 1. In on-chip memory, $\{BM_i \mid 0 \leqslant i \leqslant b\}$ are used to filter prefixes (indicating the existence of queried prefixes) at level 0 to b. $\{BF_i \mid b+1 \leqslant i \leqslant W-1\}$ are used to filter prefixes at level $b+1$ to $W-1$, while BFA_p is used to decide whether the length of LMP of a is greater than the pivot level. In off-chip memory, key-value storage are used to store FIB records ($\langle prefix, next\text{–}hop\rangle$ pairs for simplicity) and return next-hop for prefix query. Knowing that Bloom filter alone can't support deleting, we introduce counting Bloom filters [23] ($\{BFC_i \mid b+1 \leqslant i \leqslant W-1\}$ and $BFCA_p$) to assist updating for each Bloom filter and put them in off-chip saving precious on-chip memory. As shown in Figure 1, those data structures can be grouped into three functional areas, which are candidate prefix collection, key(prefix)–value(next hop) storage and probe, and assistant updating.

B. Pivot Pushing Revised

Used to initialize Bloom filters and guide updating, the expanded trie's construction from the original one is crucial in achieving memory saving and fast updating. Separating prefixes into two disjoint groups by a given prefix length can be obtained by various prefix transformation techniques. Inspired by precomputation [6] and pivot pushing [8], we carefully choose prefixes for expansion, greatly reducing memory consumption. In precomputation, for every prefix, each node in the binary search sequence ended at that prefix is checked and will be tagged by its nearest solid ancestor's next-hop (thus, is prefix expansion) if that node is a hollow. The newly tagged node is entitled *a marker* guiding IP lookup correctly. Here, in this paper, we borrow the notion of marker to denote the same meaning for nodes only in pivot level. In pivot pushing, each hollow node in the pivot level is regarded as a marker ignoring the necessity of being one. In our observation, not all the hollows must be a marker as a *hidden prefix* may be derived by that node's descendants. Specifically, considering the situation that all the addresses from that hollow node have LMPs been found in its solid descendants, in that case, we don't need to entitle that hollow (thus, is a hidden prefix).

Based on this observation, our revised pivot pushing can be achieved in three steps. First, determine whether the hollow node in the pivot level is a marker or not, then tag it with its nearest solid ancestor's next-hop if yes, and finally push the tag of each internal solid node in the pivot level into its children. To determine a marker, we recursively decide if the currently accessed node is a hidden prefix. To be a hidden prefix, the node must be a solid node or both children must be hidden prefixes, while internal node's any none exist child is considered as not a hidden prefix. Once not a hidden prefix, it is entitled *a marker* being tagged by its nearest solid ancestor's next-hop. The nearest solid ancestor is called *an expander* as it expands the next-hop to the marker. Here, we proceed to the last step. For any internal solid node with a tag in the pivot level, being it an old one or a marker, we push the tag to all the children without a tag, in which none exist child is also considered as having no tag. Note that, after the above pushing, the internal solid node becomes a hollow one and no internal solid nodes will be found in the pivot level now.

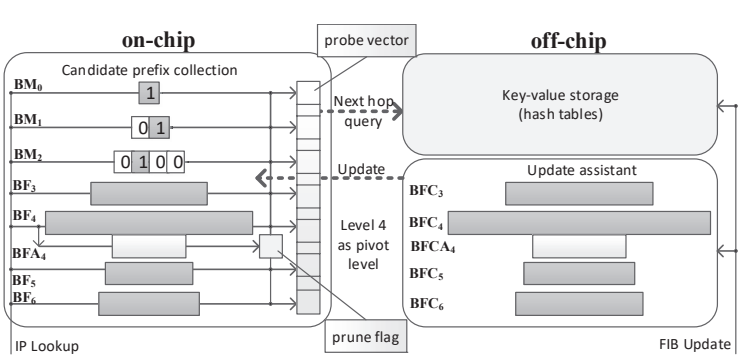

Fig. 1. Layout of the data structures.

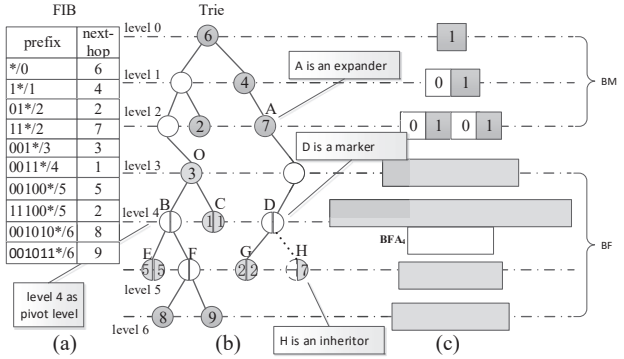

Fig. 2. Pivot pushing.

Any child that inherits the next-hop tag from its parent node is called *an inheritor*. Those nouns, *expander*, *marker*, and *inheritor*, are served as titles assigned to some nodes in the expanded trie making a better understanding of our IP update algorithm.

Applying revisited pivot pushing to the original trie, we can get an expanded one. The construction of the original trie from RIB is omitted, details can be found in [9]. Suppose we choose level $p = 4$ as pivot level as in Figure 2(b) (the choosing of an appropriate pivot level can be found in IV-B). For the convenience of illustration, we break the affected whole circle into left half and right half, representing the original node before and the modified one after pivot pushing, separately. We also draw a dotted left half circle in the place where no node existed in the original trie but a new one is created because of pivot pushing. As we can see, only nodes in the two levels, the pivot level 4 and the next level 5, may be modified.

After pivot pushing, if the place at pivot level corresponding to a has no node exist, the LMP of a can be found at some level smaller than p; if has a solid node, it can be found at level p; or has a hollow node, it can be found at a bigger level.

C. IP Lookup using Pruning

Representing a given set (such as FIB table), a Bloom filter is an m-long bit-array with k independent hash functions. When testing an element's existence, k indexes are produced by those k hash functions on it and only all 1's in the bit-array's k positions indicate it is in the set. Using Bloom filter(s) to separate prefix testing from FIB lookup, due to its false positive, a nearly constant time complexity can be achieved for IP lookup in the average case and $\Theta(W)$ in the worst case.

False positive makes Bloom filter based approaches not so efficient, and its probability f can be minimized [4] to $f = 1/2^k$ when $k = (m/n) \times ln2$, given a set of n records and a m-long bit-array. To guarantee the false positive probability no worse than f, it is enough to allocate $-2\ln f$ ($m/n = -\ln f/(\ln 2)^2 \approx -2\ln f$) bits per record using Bloom filter. To manage a set of n_i prefixes of length i, if $(-2\ln f * n_i) < 2^i$, we can use Bloom filter with $-2n\ln f$ bits to program the set saving on-chip memory, otherwise, 2^i-long bitmap can be used to represent the existence of each potential prefix since 2^i is the maximum number of available prefixes of length i. To further approaching the constant time and reducing worst case complexity while supporting fast update, we use binary search only at the pivot level. Though false positive problem can't be eliminated, pivot level pruning of candidate prefixes can greatly reduce number of probes before a hit.

In on-chip memory, we use bitmap BM_i for prefixes of length i ($0 \leqslant i \leqslant b$, b can be decided using the criteria mentioned above). Each bitmap is created to be all zero. Set $BM_i[pf] = 1$ for each tagged node pf of length i in the expanded trie. We use Bloom filter BF_i for each level i ($b+1 \leqslant i \leqslant W$). Keeping the same false positive probability $f = (1/2)^k$ with a fixed k, we allocate $m_i = -2n_i lnf$ bits for each BF_i. Set $BF_i[h_j(pf)] = 1$ for each j ($1 \leqslant j \leqslant k$, $h_j(.)$ stands for the ith hash function) for each tagged node pf of length i in the expanded trie. To support pruning at pivot level p, an additional Bloom filter BFA_p is needed. BFA_p represents the set of hollow nodes at level p in the expanded trie. The pseudo code for an IP address lookup is shown in Algorithm 1. In Stage 1, candidate prefixes are collected by testing BFA_p, $\{BF_i \mid b+1 \leqslant i \leqslant W-1\}$ and $\{BM_i \mid 0 \leqslant i \leqslant b\}$, which can be done in parallel using GPU or FPGA for further speed up. After testing, we get the prune flag and a set of candidate prefixes $\{a_{l1}, a_{l2}, \cdots, a_{lc}\}$ (all the positions $\in \{l1, l2, \cdots, lc\}$ in the probe vector are set to 1, the others 0). If the prune flag is equal to 1, it indicates that the length of LMP is bigger than p with some false positive probability, so we probe for next-hop all the candidate prefixes in a descending order of length and return immediately after a hit. If the prune flag is equal to 0, it can be asserted that the length of LMP is not bigger than p with no exception, so we only probe the candidate prefixes of lengths $\leqslant p$.

Knowing that most of the IPv4 prefixes of FIB records are at level 24, the situation that prune flag is equal to 0 occurs more frequent, reducing the number of candidate prefixes for probe and speeding up IPv4 lookup. This intuition can be used to choose an appropriate pivot level not only for IPv4 but also for IPv6 and the details can be found in IV-B.

D. Fast FIB Updating

The FIB records may change frequently as routing protocols report the network's fast-changing topology. In order to cope with highly dynamic situations, fast incremental IP update algorithm is required. In our proposed setting, updates will

Algorithm 1: IP Lookup

Input: Bitmaps: $BM_0, BM_1, \cdots, BM_b$
Input: Bloom filters: $BF_{b+1}, BF_{b+2}, \cdots, BF_W$
Input: additional Bloom filter for pivot level p: BFA_p
Input: the IP destination address: a
Output: next-hop of the longest matching prefix of a

```
// getBF(em, BF): return whether or not
   an element em is in the set stored
   by the Bloom Filter BF
// GetNextHop(pf): return a next-hop
   record from prefix-next hop storage
   given a prefix pf
// pv: probe vector, pb: prune flag
   (see Figure 1)
1  Stage 1 in the on-chip memory:
2      pb ← getBF(a_p, BFA_p);
3      if pb = 1 then tmp ← W else tmp ← p;
4      for i ← 0 to b do
5          pv [i]← BM_i[a_i];
6      for i ← b+1 to tmp do
7          pv [i]← getBF(a_i, BF_i);
8  Stage 2 in the off-chip memory:
9      for i ← tmp to b + 1 do
10         if pv[i] = 1 then
11             nextHop ← GetNextHop(a_i);
12             if nextHop is not NULL then
13                 return nextHop;
14     for i ← b to 0 do
15         if pv[i] = 1 then
16             nextHop ← GetNextHop(a_i);
17             return nextHop;
```

affect both control plane and data plane. Specifically, we first update the control plane according to the routing protocol's update message and then use the control plane's *changing information* to guide data plane's updating. Trivial updating in the control plane can be done by first update the original trie, and then construct a new expanded one by it, and the changing information can be derived by comparing the two different expanded tries. It is extremely time consuming since a wholly new expanded trie is generated. Thus, incrementally updating the expanded trie is critical in achieving fast update. Knowing that update messages can be divided into two categories: *announcement* in which a prefix with a next-hop should be added, and *withdrawal* in which an already existed prefix should be deleted, while each of those prefixes corresponds to a position/node in the original trie. In this paper, we focus only on prefixes in those messages and their impacts, and ignore others. Recall that in forming an expanded trie, whether a node at level p is a marker is determined only by its descendants, and the node and its children may be modified iff it is a marker. So we consider updating in the control plane in two different situations: when the length of the updated prefix pf is smaller than p which is the pivot level, and the others. In each situation, we separately address the two update categories.

Before exploring IP update, to facilitate fast updating, we introduce how to extend the expanded trie which was designed only for Bloom filters' initialization to *an extended trie*. Specifically, we combine the original trie and the expanded one into the extended trie keeping both information about original prefixes and expanded ones. Besides attribution of a next-hop *old tag* for each node, attributions of a *new tag*, a *title* assigned from $\{expander, marker, inheritor\}$, a *marker list* consisting of related markers, and an *expander pointer* pointing to expander are augmented to that node. Note that the value of all attributions is set to null at a node's creation. If a node has a prefix in the original trie, the next-hop of the prefix is set to the old tag of it. When a node is a marker, a pointer to its expander is set to the expander pointer of it. For each marker, a pointer to it is added to the *marker list* of its expander. When a node is an inheritor, set the new tag of it by the next-hop (new tag or old tag) of its parent. Only discovering a marker triggers trie's modification, so the extended trie is almost the same size as the expanded one.

When the length of pf is smaller than p: In this situation, as update happens above level p, any *marker* title will not be revoked and thus not be revoked the *inheritor*, but it may cause the *expander* title's hand over between different nodes which affecting some *inheritor*'s next-hop tag.

- *announcement*. If pf is already in the original trie, we call it *renew*, otherwise *add*.
  To renew pf, first set the pf node's old tag by the next-hop of pf and then check if it is an expander. If yes, do *expander renew* for pf node. Specifically, for each marker in the *marker list* set the new tag of the marker's inheritor children by the expander's next-hop old tag.
  To add pf, first create a new node for pf in the trie if no node in that position of pf, set the old tag of the pf node by the next-hop of pf, and then up tracing its ancestors to find an expander. Once been found, first extract markers having prefix pf from the *marker list* and add them to the *marker list* of the pf node which is now set to be an expander, and then do *expander renew* for pf node. Revoke the expander node's title once the *marker list* becomes null.
- *withdrawal*. To withdrawal pf, first set the old tag and the title by null, delete the pf node if it is a leaf, and then check if the pf node is an expander. If yes, up tracing its ancestors to find a solid node. Once been found, first extract each marker from the *marker list* of the pf node and add them to the ancestor, and then set the ancestor to be an expander if it is not and do *expander renew* for the ancestor with those newly added markers.

When the length of pf is not smaller than p: As whether a node at level p is a marker is determined only by its descendants, in this situation, we inspect the *marker* title's changing at level p when updating a prefix of length $\geqslant p$.

- *announcement*. To renew pf, just set the pf node's old tag by the next-hop of pf. To add pf, first create a new node for pf if no node in that position of pf, set the old tag of the pf node by the next-hop of pf, revoke the *inheritor* title if the pf node was an inheritor before, and then inspect the title's changing of its ancestor at level p. The procedure for determining if the ancestor after pf's adding is a marker can be found in Subsection III-B. Four cases may be happen, but only two of them need be considered carefully. If the ancestor's title stays still, do nothing. If the ancestor is now changed to be a marker, it must be that the ancestor's position has no node before pf's adding and a newly added leaf pf node forces the hollow ancestor node's creation, and obviously the ancestor must be a marker. In this case, do *marker creation* for the ancestor in the extended trie. Note that *marker creation* is the same as *revised pivot pushing*'s last two steps, so we omit it for space saving. If the title of the ancestor is revoked, do *marker revocation* for it. Specifically, delete the marker from the *marker list* of the expander node pointed by the ancestor's *expander pointer*, revoke the expander node's title once the *marker list* becomes null, and then set the title of the ancestor and its children by null and delete any child node if it is a leaf and it's old tag is null.
- *withdrawal*. To withdrawal pf, first set the old tag and the title by null, delete the pf node if it is a leaf, and then inspect the title's changing of its ancestor at level p. Like add pf, four cases may be happen, and only two of them need rigorous treatment. If the ancestor's title stays still, do nothing. If the ancestor is now changed to be a marker, do *marker creation* for it. If the title of the ancestor is revoked, it must be that it has no next-hop old tag and has only one descendant which is the pf node in the original trie. Do *marker revocation* for that ancestor and delete it.

IV. Experimental Results

A. Data Sets

In our experiments, we focus on the LPM with IPv4 where $W = 32$. We use IPv4 FIBs downloaded from 5 routers provided by RIPE [24] in June 8 2014, and query for each FIB by IP packet traffics containing approximately 5 million IP packets. The total numbers of prefixes in the 5 FIBs are 498409, 518156, 457519, 500703, 548596, respectively. To test the robustness and efficiency of our proposed IP lookup algorithm, traffics generated meet one of the two patterns:

- random traffic: randomly generated IP addresses.
- prefix-based traffic: IP addresses generated obey the prefix distribution of the corresponding FIB.

B. Pivot Level Choosing

Some consideration on how to choose an appropriate pivot level is given before evaluating the performance of our pruning algorithm.We calculate the proportion of prefixes at each level to the total prefixes for each of the 5 FIBs [24] we collected and get the mean and the standard deviation of those 5 proportions at each level (as shown in Figure 3). As we can see from Figure 3, length-24 prefixes take up more than half of the total number and prefixes of length less than 8 hardly exists. Considering that pivot pushing may change the number of prefixes at level p and $p+1$ and our LPMP uses an additional Bloom filter BFA_p for pruning, to keep the false positive probability unchanged, additional on-chip memory should be allocated after pivot pushing. Figure 4 shows the additional on-chip memory needed at some level if using pivot pushing at that level. If choosing level 24 as pivot level, from Figure 3 and 4, pivot pushing will need 0.9% additional on-chip memory while 52.3% of the prefixes are at this level. More prefixes at some level, with higher probability the LPM is at that level; Less increment in on-chip memory, less pressure for on-chip memory usage. So level 24 is the best choice to meet the two requirement at the same time.

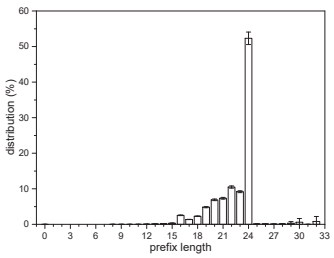

Fig. 3. Mean and the standard deviation of the prefix length distribution for the 5 FIBs.

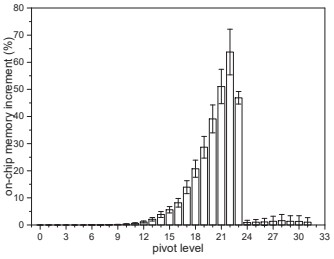

Fig. 4. Mean and the standard deviation of the on-chip memory increment after pivot pushing for the 5 FIBs.

To test the performance for different pivot levels, we fix the on-chip memory space of 6.144Mbit, calculate the average number of probes after pivot pushing at different levels for the two traffic patterns, and plot the two curves in Figure 5. It shows that different pivot level can affect the average number of probes. For prefix based traffic, we get the least three average number of probes of 1.011, 1.008, 1.004 at level 26, 25 and 24 respectively; for random traffic, 1.051, 1.045, 1.046 at level 24, 13, and 12. Choosing level 24 as pivot level can always get the least average number of probes for the two traffic patterns. Below we choose level 24 as our pivot level for performance evaluation.

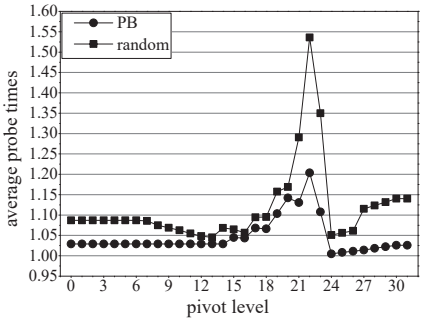

Fig. 5. Avg. #probes after pivot pushing for the two traffic patterns.

C. Evaluation Results

We evaluated our IP lookup algorithm by taking into consideration the relationship between number of probes per IP lookup and the total on-chip memory consumption. As our work is an extension of the well-know IP lookup algorithm — PBF who first introduce Bloom filter to assist longest prefix

matching, we compared the two algorithms using the 5 FIBs with random traffics and prefix-based traffics, separately.

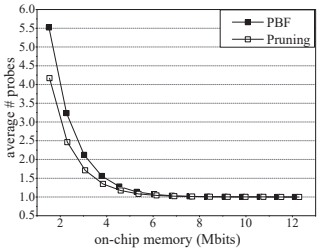

Fig. 6. Avg. #probes vs. memory sizes using random traffics.

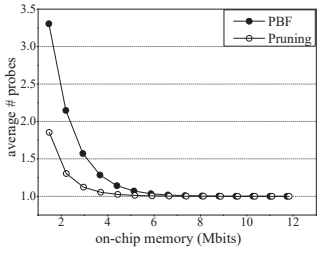

Fig. 7. Avg. #probes vs. memory sizes using prefix-based traffics.

Average # probes using random traffics: *Experimental results show that for random traffics, to meet an average # probes less than 1.003, our algorithm saves 7.2% on-chip memory space than PBF.* As approximately 30% IPs in the traffic didn't hit prefixes in FIB, on-chip memory saving is not obvious. If the number of none matching IP packets is relatively high, more on-chip memory is needed to keep the average # probes unchanged or average # probes will be large. Using random traffics, Figure 6 reports the mean of the 5 experiment's average # probes given different on-chip memory space. As we can see, the average # probes for our algorithm are always lower than that of PBF. To achieve an average # probes less than 1.003, our algorithm needs 8.715 Mbits while PBF needs 9.391 Mbits to represent FIB of approximately 500 thousand prefix records.

Average # probes using prefix-based traffics: *Experimental results show that for prefix-based traffics, to meet an average # probes less than 1.003, our algorithm saves 24.7% on-chip memory space than PBF.* Using prefix-based traffics, Figure 7 plots the mean of the 5 experiment's average # probes given different on-chip memory space. As we can see, the average # probes for our algorithm are always lower than that of PBF and our algorithm converges to an average # probes equals 1 sharply. To achieve an average # probes less than 1.003, our algorithm needs 6.495 Mbits while PBF needs 8.628 Mbits to represent FIB of approximately 500 thousand prefix records.

V. Conclusion

IP lookup is a classic but still a hot issue nowadays [8], [25] due to the explosive growth of backbone FIBs. In this paper, we propose a novel IP lookup algorithm, named Longest Prefix Matching with Pruning. Our algorithm uses prefix Bloom filters [4] to represent the trie after applying revised pivot pushing. The key improvement of our paper is that we build a Bloom filter for the hollow nodes at the pivot level to reduce #probes. While achieving the same average #probes of 1.003 as that of PBF, $7\%-24\%$ on-chip memory space can be saved.

Acknowledgment

This research was supported in part by the National Key Research and Development Program of China No. 2017YFB1400700, and the National Natural Science Foundation of China under Grant 61672061, 61571352 and U1736216.

References

[1] "Source code of Longest Prefix Matching with Pruning," https://github.com/hu-jun/LPMwithPruning.
[2] "BGP routing table analysis reports," http://bgp.potaroo.net/.
[3] F. Wang and M. Hamdi, "Matching the speed gap between sram and dram," in *High Performance Switching and Routing, 2008. International Conference on*. IEEE, 2008, pp. 104–109.
[4] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 201–212.
[5] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On adding bloom filters to longest prefix matching algorithms," *Computers, IEEE Transactions on*, vol. 63, no. 2, pp. 411–423, 2014.
[6] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, *Scalable high speed IP routing lookups*. ACM, 1997, vol. 27, no. 4.
[7] J. H. Mun and H. Lim, "New approach for efficient ip address lookup using a bloom filter in trie-based algorithms," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1558–1565, 2016.
[8] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP lookup performance with FIB explosion," in *ACM SIGCOMM Computer Communication Review*, vol. 44, 2014, pp. 39–50.
[9] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *Network, IEEE*, vol. 15, no. 2, pp. 8–23, 2001.
[10] K. Sklower, "A tree-based packet routing table for berkeley unix." in *USENIX Winter*, vol. 1991, 1991, pp. 93–99.
[11] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 1, pp. 1–40, 1999.
[12] P. Crescenzi, L. Dardini, and R. Grossi, "Ip address lookupmade fast and simple," in *Algorithms-ESA99*. Springer, 1999, pp. 65–76.
[13] S. Suri, G. Varghese, and P. R. Warkhede, "Multiway range trees: Scalable ip lookup with fast updates," in *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, vol. 3, 2001, pp. 1610–1614.
[14] F. Zane, G. Narlikar, and A. Basu, "Coolcams: Power-efficient tcams for forwarding engines," in *IEEE INFOCOM 2003.*, vol. 1, pp. 42–52.
[15] K. Zheng, C. Hu, H. Lu, and B. Liu, "A tcam-based distributed parallel ip lookup scheme and performance analysis," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 4, pp. 863–875, 2006.
[16] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for ip address lookup," in *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pp. 81–90.
[17] H. Le, W. Jiang, and V. K. Prasanna, "A sram-based architecture for trie-based ip lookup using fpga," in *IEEE Field-Programmable Custom Computing Machines, 2008. 16th International Symposium on*, pp. 33–42.
[18] D. Pao, Z. Lu, and Y. H. Poon, "Ip address lookup using bit-shuffled trie," *Computer Communications*, vol. 47, pp. 51–64, 2014.
[19] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.
[20] J. Zhao, X. Zhang, X. Wang, and X. Xue, "Achieving o (1) ip lookup on gpu-based software routers," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 429–430, 2010.
[21] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM'98*, pp. 1240–1247.
[22] A. Lucchesi, A. C. Drummond, and G. Teodoro, "High-performance IP lookup using intel xeon phi: a bloom filters based approach," *J. Internet Services and Applications*, vol. 9, no. 1, pp. 3:1–3:18, 2018.
[23] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.
[24] "RIPE Network Coordination Centre," https://www.ripe.net/.
[25] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," in *Proc. ACM SIGCOMM*, 2015.