# Constant IP Lookup With FIB Explosion

Tong Yang, Gaogang Xie, Alex X. Liu, Qiaobin Fu, Yanbiao Li, Xiaoming Li, and Laurent Mathy

***Abstract*— With the fast development of Internet, the forwarding tables in backbone routers have been growing fast in size. An ideal IP lookup algorithm should achieve constant, yet small, IP lookup time, and on-chip memory usage. However, no prior IP lookup algorithm achieves both requirements at the same time. In this paper, we first propose SAIL, a splitting approach to IP lookup. One splitting is along the dimension of the lookup process, namely finding the prefix length and finding the next hop, and another splitting is along the dimension of prefix length, namely IP lookup on prefixes of length less than or equal to 24 and that longer than 24. Second, we propose a suite of algorithms for IP lookup based on our SAIL framework. Third, we implemented our algorithms on four platforms: CPU, FPGA, GPU, and many-core. We conducted extensive experiments to evaluate our algorithms using real FIBs and real traffic from a major ISP in China. Experimental results show that our SAIL algorithms are much faster than well known IP lookup algorithms.**

***Index Terms*— IP lookup, FIB, SAIL, constant, IPv6.**

## I. Introduction

### A. Background and Motivation

WITH the fast development of Internet, the size of Fowarding Information Base FIB (FIB) in backbone routers grows rapidly. According to the RIPE Network Coordination Centre, FIB sizes have become more than 700,000 entries [3]. At the same time, cloud computing and network applications have driven the expectation on router throughput to the scale of 200 Gbps. The fast growth of FIB sizes and throughput demands bring significant challenges to IP lookup (*i.e.*, FIB lookup). An ideal IP lookup algorithm should satisfy the following two harsh requirements. First, *IP lookup time should meet wire speed yet remain constant as FIB sizes grow.* IP lookup time is per packet cost and should be optimized to the extreme to meet wire speed. Second, *on-chip memory usage should meet capacity constraints yet remain constant as FIB sizes grow*. On-chip memory (such as CPU cache and FPGA block RAM) is about 10 times faster than off-chip DRAM [13], but is limited in size (in the scale of a few MB) and much more expensive than DRAM; furthermore, as on-chip memory technologies advance, its sizes do not grow much as compared to DRAM. With network virtualization, a virtual router could have hundreds of virtual FIBs, which makes fast FIB lookup with small on-chip memory even more critical. Without satisfying these requirements, router performance will degrade as FIB grows, and router hardware will have to be upgraded periodically.

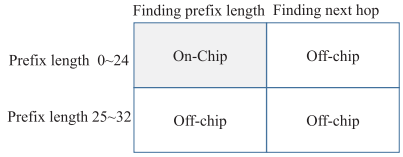


Fig. 1. Two-dimensional splitting of IP lookup.

### B. Summary and Limitations of Prior Art

IP lookup has long been a core networking issue and various schemes have been proposed. However, none of them satisfies the two requirements of both constant lookup time and constant on-chip memory usage. Some algorithms can achieve constant IP lookup time, such as TCAM based schemes [15], [25] and FPGA based schemes [19], [21], but their on-chip memory usage will grow quickly as FIB sizes grow. Some algorithms, such as full-expansion [41] and DIR-24-8 [18], can achieve constant memory usage by simply pushing all prefixes to levels 24 and 32, but even the lookup table for level 24 alone is too large to be stored in on-chip memory.

### C. Proposed SAIL Approach

In this paper, we propose SAIL, a Splitting Approach to IP Lookup. We split the IP lookup problem along two dimensions as illustrated in Figure 1. First, we split the IP lookup problem into two sub-problems along the dimension of the lookup process: finding the prefix length (*i.e.*, finding the length of the longest prefix that matches the given IP address) and finding the next hop (*i.e.*, finding the next hop of this longest matched prefix). This splitting gives us the opportunity of solving the prefix length problem in on-chip memory and the next hop problem in off-chip memory. Furthermore, since on-chip and off-chip memory are two entities, this splitting allows us to potentially pipeline the processes of finding the prefix length and the next hop.

iManuscript received September 25, 2017; revised April 13, 2018; accepted June 6, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Schapira. This work was supported in part by the Primary Research & Development Plan of China under Grant 2016YFB1000304, in part by the National Basic Research Program of China (973 Program) under Grant 2014CB340405, in part by the NSFC under Grant 61672061, Grant 61472184, and Grant 61321491, in part by the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, National Science Foundation under Grants CNS 1345307, CNS 1616317, and CNS 1616273, and in part by the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. The preliminary version of this paper titled "Guarantee IP Lookup Performance with FIB Explosion" was published in SIGCOMM 2014 [61]. *(Corresponding authors: Gaogang Xie; Alex X. Liu.)*

T. Yang and X. Li are with the Department of Computer and Science, Peking University, Beijing 100871, China (e-mail: yangtongemail@gmail.com).

G. Xie is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: xie@ict.ac.cn).

A. X. Liu is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: alexliu@cse.msu.edu).

Q. Fu is with the Department of Computer Science, Boston University, Boston, MA 02215 USA.

Y. Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China.

L. Mathy is with the Department of Computer and Science, University of Liége, 4000 Liege, Belgium (e-mail: laurent.mathy@ulg.ac.be).

Digital Object Identifier 10.1109/TNET.2018.2853575

Second, we split the IP lookup problem into two sub-problems along the dimension of prefix length: length $\leqslant 24$ and length $\geqslant 25$. This splitting is based on our key observation that on backbone routers, for almost all traffic, the longest matching prefix has a length $\leqslant 24$. This intuitively makes sense because typically backbone routers do not directly connect to small subnets whose prefixes are longer than 24. Our observation may not hold for edge routers, and the edge routers could have more long prefixes. However, even if all prefixes are longer prefixes, the worst case performance of our algorithm is still 2 off-chip memory accesses, and the on-chip memory usage is still bounded $\leqslant$ 4MB (See Table II). The scope of this paper is on backbone routers. The key benefit of this splitting is that we can focus on optimizing the IP lookup performance for traffic whose longest matching prefix length is $\leqslant 24$.

There is some prior work that performed splitting along the dimension of the lookup process or the dimension of prefix length; however, no existing work performed splitting along both dimensions. Dharmapurikar *et al.* proposed to split the IP lookup process into two sub-problems: finding the prefix length using Bloom filters and finding the next hop using hash tables [47]. Gupta *et al.* [18] and Pierluigi *et al.* [41] proposed to split IP prefixes into 24 and 32 based on the observation that $99.93\%$ of the prefixes in a backbone router FIB has a length of less than or equal to 24. Note that our IP prefix splitting criteria is different because our splitting is based on traffic distribution and their splitting is based on prefix distribution.

### D. Technical Challenges and Solutions

The first technical challenge is to *achieve constant, yet small, on-chip memory usage for any FIB size*. To address this challenge, in this paper, we propose to find the longest matching prefix length for a given IP address using bitmaps. Given a set of prefixes in a FIB, for each prefix length $i$ $(0 \leqslant i \leqslant 32)$, we first build a $B_i[0..2^i - 1]$ whose initial values are all 0s. Then, for each prefix $p$, we let $B_i[|p|] = 1$ where $|p|$ denotes the binary value of the first $i$ bits of $p$. Thus, for all prefixes of lengths 0$\sim$24 in any FIB, the total memory size for all bitmaps is $\sum_{i=0}^{24} 2^i = 4$MB, which is small enough to be stored in on-chip memory.

The second technical challenge is to *achieve constant, yet small, IP lookup time for any FIB size*. To address this challenge, in this paper, we classify the prefixes in the given FIB into two categories: those with length $\leqslant 24$ and those with length $\geqslant 25$. (1) For prefixes of length $\leqslant 24$, for each prefix length $i$ $(0 \leqslant i \leqslant 24)$, we build a next hop array $N_i[0..2^i - 1]$ in off-chip memory so that for each prefix $p$ whose next hop is $n(p)$, we let $N_i[|p|] = n(p)$. Thus, given an IP address $\boldsymbol{a}$, we first find its prefix length using bitmaps in on-chip memory and find the next hop using one array lookup in off-chip memory. To find the prefix length using bitmps, for $i$ from 24 to 0, we test whether $B_i[\boldsymbol{a} \gg (32 - i)] = 1$; once we find the first $i$ so that $B_i[\boldsymbol{a} \gg (32-i)] = 1$ holds, we know that the longest matching prefix length is $i$. Here $\boldsymbol{a} \gg (32 - i)$ means right shifting $\boldsymbol{a}$ by $32 - i$ bits. In this step, the maximum number of on-chip memory accesses is 25. To find the next hop, suppose the longest matching prefix length for $\boldsymbol{a}$ is $i$, we can find its next hop $N_i[\boldsymbol{a} \gg (32-i)]$ by one off-chip memory access. (2) For prefixes of length $\geqslant 25$, many IP lookup schemes can be plugged into our SAIL framework. Possible schemes include TCAM (Ternary Content Addressable Memory), hash tables, and next-hop arrays. Choosing which scheme to deal with prefixes of length $\geqslant 25$ depends on design priorities, but have little impact on the overall IP lookup performance because most traffic hits prefixes of length $\leqslant 24$. For example, to bound the worst case lookup speed, we can use TCAM or next hop arrays. For next hop arrays, we can expand all prefixes of length between 25 and 31 to be 32, and then build a chunk ID (*i.e.*, offsets) array and a next hop array. Thus, the worst case lookup speed is two off-chip memory accesses.

The third technical challenge is to *handle multiple FIBs for virtual routers* with two even harsher requirements: (1) Multi-FIB lookup time should meet wire speed yet remain constant as FIB sizes and FIB numbers grow. (2) On-chip memory usage should meet capacity constraints yet remain constant as FIB sizes and FIB numbers grow. To address this challenge, we overlay all FIB tries together so that all FIBs have the same bitmaps; furthermore, we overlay all next hop arrays together so that by the next hop index and the FIB index, we can uniquely locate the final next hop.

## II. Related Work

As IP lookup is a core networking issue, much work has been done to improve its performance. We can categorize prior work into trie-based algorithms, Bloom filter based algorithms, range-based algorithms, TCAM-based algorithms, FPGA-based algorithms, GPU-based algorithms, and multi-FIB lookup algorithms.

*Trie-Based Algorithms:* Trie-based algorithms use the trie structure directly as the lookup data structure or indirectly as an auxiliary data structure to facilitate IP lookup or FIB update. Example algorithms include binary trie [36], path-compressed trie [26], k-stride multibit trie [50], full expansion/compression [41], LC-trie [49], Tree Bitmap [52], priority trie [23], Lulea [37], DIR-24-8 [18], flashtrie [34], shapeGraph [20], trie-folding [16], DPP [28], OET [22], DTBM [45], multi-stride compressed trie [38], and poptrie [7]. More algorithm can be found in the literature [29], [36], [57], [60], [62].

*Bloom Filter Based Algorithms:* Dharmapurikar *et al.* proposed the PBF algorithm where they use Bloom filters to first find the longest matching prefix length in on-chip memory and then use hash tables in off-chip memory to find the next hop [47]. Lim *et al.* [30] proposed to use one bloom filter to find the longest matching prefix length. These Bloom filter based IP lookup algorithms cannot achieve constant lookup time because of false positives and hash collisions. Furthermore, to keep the same false positive rate, their on-chip memory sizes grow linearly with the increase of FIB size. Bloom filter variants can be found in the literature [8]–[10], [58], [59].

*Range-Based Algorithms:* Range-based algorithms are based on the observation that each prefix can be mapped into a range in level 32 of the trie. Example such algorithms are binary search on prefix lengths [32], binary range search [36], multiway range trees [42], and DXR [33].

*TCAM-Based Algorithms:* TCAM can compare an incoming IP address with all stored prefixes in parallel in hardware using one cycle, and thus can achieve constant lookup time. However, TCAM has very limited size (typically a few Mbs like on-chip memory sizes), consumes a huge amount of power due to the parallel search capability, generates a lot

TABLE I
SYMBOLS USED IN THE PAPER

| Symbol | Description |
|---|---|
| $B_i$ | array for level $i$ |
| $N_i$ | next hop array for level $i$ |
| $C_i$ | chunk ID array for level $i$ |
| $BN_i$ | combined array of $B_i$ and $N_i$ |
| $BCN_i$ | combined array of $B_i, C_i$ and $N_i$ |
| $\boldsymbol{a}$ | IP address |
| $v$ | trie node |
| $p$ | prefix |
| $\|p\|$ | value of the binary string in prefix $p$ |
| $p(v)$ | prefix represented by node $v$ |
| $n(v)$ | next hop of solid node $v$ |
| $l$ | trie level |
| $\boldsymbol{a}_{(i,j)}$ | integer value of bit string of $\boldsymbol{a}$ from $i$-th bit to $j$-th bit |

of heat, is very expensive, and difficult to update. Some schemes have been proposed to reduce power consumption by enabling only a few TCAM banks to participate in each lookup [15]. Some schemes use multiple TCAM chips to improve lookup speed [25], [39], [44]. Devavrat and Pankaj proposed to reduce the movement of prefixes for fast updating [12].

*FPGA-Based Algorithms:* There are two main issues to address for FPGA-based IP lookup algorithms: (1) how to store the whole FIB in on-chip memory, and (2) how to construct pipeline stages. Some early FPGA-based algorithms proposed to construct compact data structures in on-chip memory [31], [43]; however, these compact data structures make the lookup process complex and therefore increase the complexity of FPGA logics. For FPGA in general, the more complex the logics are, the lower the clock frequency will be. To improve lookup speed, Hamid *et al.* proposed to only store a part of data structure in on-chip memory using hashing [19]. To balance stage sizes, some schemes have been proposed to adjust the trie structure by rotating some branches or exchanging some bits of the prefixes [11], [14], [21].

*GPU-Based Algorithms:* Leveraging the massive parallel processing capability of GPU, some schemes have been proposed to use GPU to accelerate IP lookup [46], [63].

*Multi-FIB Lookup Algorithms:* The virtual router capability has been widely supported by commercial routers. A key issue in virtual routers is to perform IP lookup with multiple FIBs using limited on-chip memory. Several schemes have been proposed to compress multiple FIBs [24], [27], [48].

## III. SAIL BASICS

In this section, we present the basic version of our SAIL algorithms. Table I lists the symbols used in this paper.

### A. Splitting Lookup Process

We now describe how we split the lookup process for a given IP address into the two steps of finding its longest matching prefix length and finding the next hop. Given a FIB table, we first construct a trie. An example trie is in Figure 2(b). Based on whether a node represents a prefix in the FIB, there are two types of nodes: *solid nodes* and *empty nodes*. A node is solid if and only if the prefix represented by the node is in the FIB. That is, a node is solid if and only if it has a next hop. A node is an empty node if and only if it has no next hop. Each solid node has a label denoting the next hop of the prefix represented by the node. For any

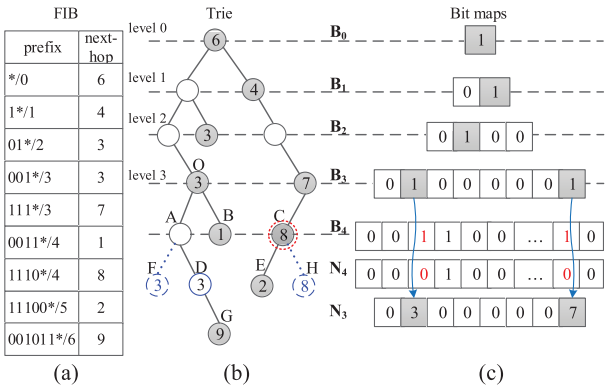


Fig. 2. Basic SAIL Algorithm. (a) is an FIB. (b) is the corresponding trie. (c) includes bitmaps and next hop arrays.

node, its distance to the root is called its *level*. The level of a node locates a node vertically. Any trie constructed from a FIB has 33 levels. For each level $i$, we construct a array $B_i[0..2^i-1]$ of length $2^i$, and the initial values are all $0s$. At each level $i$ ($0 \leqslant i \leqslant 32$), for each node $v$, let $p(v)$ denote its corresponding prefix and $|p(v)|$ denote the value of the binary string part of the prefix (*e.g.*, $|11*| = 3$). If $v$ is solid, then we assign $B_i[|p(v)|] = 1$; otherwise, we assign $B_i[|p(v)|] = 0$. Here $|p(v)|$ indicates the horizontal position of node $v$ because if the trie is a complete binary tree then $v$ is the $|p(v)|$-th node at level $i$. Figure 2(c) shows the s for levels 0 to 4 of the trie in 2(b). Taking $B_3$ for level 3 as an example, for the two solid nodes corresponding to prefixes *001*/3* and *111*/3*, we have $B_3[1] = 1$ and $B_3[7] = 1$. Given the 33 s $B_0, B_1, \cdots, B_{32}$ that we constructed from the trie for a FIB, for any given IP address $\boldsymbol{a}$, for $i$ from 32 to 0, we test whether $B_i[\boldsymbol{a} \gg (32-i)] = 1$; once we find the first $i$ that $B_i[\boldsymbol{a} \gg (32-i)] = 1$ holds, we know that the longest matching prefix length is $i$. Here $\boldsymbol{a} \gg (32-i)$ means right shifting $\boldsymbol{a}$ by $32-i$ bits. For each $B_i$, we construct a next hop array $N_i[0..2^i-1]$, whose initial values are all 0s. At each level $i$, for each prefix $p$ of length $i$ in the FIB, denoting the next hop of prefix $p$ by $n(p)$, we assign $N_i[|p|] = n(p)$. Thus, for any given IP address $\boldsymbol{a}$, once we find its longest matching prefix length $i$, then we know its next hop is $N_i[\boldsymbol{a} \gg (32-i)]$.

### B. Splitting Prefix Length

Based on our observation that almost all the traffic of backbone routers has the longest matching prefix length $\leqslant 24$, we split all prefixes in the given FIB into prefixes of length $\leqslant 24$, which we call *short prefixes* and prefixes of length $\geqslant 25$, which we call *long prefixes*. By this splitting, we want to store the s of prefixes of length $\leqslant 24$ in on-chip memory. However, given an IP address, because it may match a long prefix, it seems that we need to search among both short and long prefixes, which makes this splitting not much useful. In this paper, we propose a technique called *pivot pushing* to address this issue. Our basic strategy is that for a given IP address, we first test its longest matching prefix length is within $[0, 24]$ or $[25, 32]$; thus, after this testing, we continue to search among either short prefixes or long prefixes, but not both. We call level 24 the *pivot level*.

Given a trie and a pivot level, the basic idea of pivot pushing is two-fold. First, for each internal solid node on the pivot level, we push its label (*i.e.*, the next hop) to a level below the pivot level. Second, for each internal empty nodes on the pivot

level, we let it inherit the label of its nearest solid ancestor node, *i.e.*, the next hop of the first solid node along the path from this empty node to the root, and then push this inherited label to a level below the pivot level. In this paper, we assume that the root always has a label, which is the default next hop. Thus, for any internal empty nodes on the pivot level, it always can inherit a label.

Given a trie and an IP address $\boldsymbol{a}$, traversing $\boldsymbol{a}$ from the root of the trie downward, for any internal or leaf node $v$ that the traversal passes, we say $\boldsymbol{a}$ *passes* $v$. Based on the above concepts, we introduce Theorem 1.

*Theorem 1: Given a trie constructed from a FIB, after pivot pushing, for any IP address $\boldsymbol{a}$, $\boldsymbol{a}$ passes a node on the pivot level if and only if its longest matching prefix is on the pivot level or below.*

*Proof:* Given a trie and an IP address $\boldsymbol{a}$ that passes a node $v$ on the pivot level, there are two cases: (1) $v$ is a leaf node, and (2) $v$ is an internal node. For the first case where $v$ is a leaf node, then $\boldsymbol{a}$'s longest matching prefix is $p(v)$ (*i.e.*, the prefix represented by node $v$) and thus $\boldsymbol{a}$'s longest matching prefix is on the pivot level. For the second case where $v$ is an internal node, because of pivot pushing, $\boldsymbol{a}$ must pass a solid node on a level below the pivot level, which means that $\boldsymbol{a}$'s longest matching prefix is below the pivot level. □

Based on Theorem 1, we construct the bitmap array for the pivot level $l$ as follows: for any node $v$ at level $l$, we assign $B_l[|p(v)|] = 1$; in other words, $B_l[i] = 0$ if and only if there is no node at level $l$ that corresponds to the prefix denoted by $i$. Thus, given an IP address $\boldsymbol{a}$, $B_l[\boldsymbol{a} \gg (32-l)] = 1$ if and only if its longest matching prefix is on the pivot level or below. In SAIL, we choose level 24 to be the pivot level. By checking whether $B_{24}[\boldsymbol{a} \gg 8] = 1$, we know whether the longest matching prefix length is $\leqslant 23$ or $\geqslant 24$, which will guide us to search either up or down. Consider the example in Figure 2(b), taking level 4 as the pivot level, node C at level 4 is an internal solid node, pushing C to level 5 results in a new leaf solid node H with the same next hop as C. Note that after pushing node C down, node C becomes empty.

Given a pivot pushed trie, we build a bitmap array and a next hop array for each level of 0 to 24 as above. Note that for any $i$ ($0 \leqslant i \leqslant 23$) and any $j$ ($0 \leqslant j \leqslant 2^i - 1$), $B_i[j] = 1$ if and only if there is a solid node at level $i$ that corresponds to the prefix denoted by $j$; for level 24 and any $j$ ($0 \leqslant j \leqslant 2^{24} - 1$), $B_{24}[j] = 1$ if and only if there is a node, no matter solid or empty, at level 24 that corresponds to the prefix denoted by $j$. Note that $B_{24}[j] = 1$ and $N_{24}[j] > 0$ if and only if there is a leaf node at level 24 that corresponds to the prefix denoted by $j$, which means that the longest matching prefix length is 24. Note that $B_{24}[j] = 1$ and $N_{24}[j] = 0$ if and only if there is an empty node at level that corresponds to the prefix denoted by $j$, which means that the longest matching prefix length is $\geqslant 25$. Thus, given an IP address $\boldsymbol{a}$, if $B_{24}[\boldsymbol{a} \gg 8] = 0$, then we further check whether $B_{23}[\boldsymbol{a} \gg 9] = 1$, $B_{22}[\boldsymbol{a} \gg 10] = 1$, $\cdots$, $B_0[\boldsymbol{a} \gg 32] = 1$ until we find the first 1; if $B_{24}[\boldsymbol{a} \gg 8] = 1$, then we know $\boldsymbol{a}$'s longest matching prefix length is $\geqslant 24$ and further lookup its next hop in off-chip memory. It is easy to compute that the on-chip memory usage is fixed as $\sum_{i=0}^{24} 2^i = 4\ MB$. Consider the example in Figure 2. Given an address *001010*, as the pivot level is 4, since $B_4[001010 \gg 2] = B_4[0010] = B_4[2] = 1$ and $N_4[001010 \gg 2] = N_4[0010] = N_4[2] = 0$, then we know that the longest matching prefix length is longer than 4 and we will continue the search at levels below 4.

The pseudocode for the above SAIL Basic, denoted by SAIL_B, is shown in Algorithm 1.

**Algorithm 1 SAIL_B**

**Input**: arrays: $B_0, B_1, \cdots, B_{24}$
**Input**: Next hop arrays: $N_0, N_1, \cdots, N_{24}$
**Input**: $\boldsymbol{a}$: an IP address
**Output**: next hop of the longest matched prefix

```
if B_24[a ≫ 8] = 0 then
    for j = 23; j > 0; j−− do
        if B_j[a ≫ (32 − j)] = 1 then
            return N_j[a ≫ (32 − j)]
        end
    end
end
else if N_24[a ≫ 8] > 0 then
    return N_24[a ≫ 8]
end
else
    lookup at levels 25 ∼ 32
end
```

There are multiple ways to handle prefixes of length $\geqslant 25$. Below we give one simple implementation using next hop arrays. Let the number of internal nodes at level 24 be $n$. We can push all solid nodes at levels 25∼31 to level 32. Afterwards, the number of nodes at level 32 is $256*n$ because each internal node at level 24 has a complete subtree with 256 leaf nodes, each of which is called a *chunk*. As typically $256*n$ is much smaller than $2^{32}$ based on our experimental results on real FIBs, constructing a next hop array of size $2^{32}$ wastes too much memory; thus, we construct a next hop array $N_{32}$ of size $256*n$ for level 32. As we push all solid nodes at levels from 25 to 31 to level 32, we do not need s $B_{25}, B_{26}, \cdots, B_{32}$. Now consider the nodes at level 24. For each leaf node, its corresponding entry in $B_{24}$ is 1 and its corresponding entry in next hop array $N_{24}$ is the next hop of this node. For each internal node, its corresponding entry in $B_{24}$ is 1 and its corresponding entry in next hop array $N_{24}$ is the chunk ID in $N_{32}$, multiplying which by 256 plus the last 8 bits of the given IP address locates the next hop in $N_{32}$. To distinguish these two cases, we let the next hop be a positive number and the chunk ID to be a negative number whose absolute value is the real chunk ID value. To have negative values, chunk IDs are named starting from 1. With our pivot pushing technique, looking up an IP address $\boldsymbol{a}$ is simple: if $B_{24}[\boldsymbol{a} \gg 9] = 0$, then we know the longest matching prefix length is within $[0, 23]$ and further test whether $B_{23}[\boldsymbol{a} \gg 8] = 1$; if $B_{24}[\boldsymbol{a} \gg 8] = 1 \wedge N_{24}[\boldsymbol{a} \gg 8] > 0$, then we know that the longest matching prefix length is 24 and the next hop is $N_{24}[\boldsymbol{a} \gg 8]$; if $B_{24}[\boldsymbol{a} \gg 8] = 1 \wedge N_{24}[\boldsymbol{a} \gg 8] < 0$, then we know that the longest matching prefix length is longer than 24 and the next hop is $N_{32}[(|N_{24}[\boldsymbol{a} \gg 8]|-1)*256+(\boldsymbol{a}\&255)]$.

### C. FIB Update for SAIL Basic

We now discuss how to adjust the lookup data structures when the given FIB changes. Note that the FIB update performance for levels 25∼32 is less critical than that for levels 0∼24. As most traffic hits levels 0∼24, when the lookup

data structures for levels 0∼24 in on-chip memory change, no lookup can be performed before changes are finished and therefore may cause packet losses. For the lookup data structures in off-chip memory, one possible solution is to store two copies of the lookup data structures in two memory chips so that while one copy is being updated, the other copy can be used to perform lookups. Furthermore, many IP lookup schemes that can handle prefixes of length $\geqslant 25$ can be plugged into SAIL_B. Different IP lookup schemes have different FIB update algorithms. Therefore, in this paper, we focus on the update of data structures in on-chip memory.

For SAIL_B, updating the on-chip lookup data structures is simple: given an update prefix $p$ with length of $l$, whose next hop is denoted by $h$ where $h = 0$ means to withdraw prefix $p$ and $h > 0$ means to announce prefix $p$, if $l < 24$, we assign $B_l[|p|] = (h > 0)$ (*i.e.*, if $h > 0$, then we assign $B_l[|p|] = 1$; otherwise, we assign $B_l[|p|] = 0$). If $l = 24$, for the same update, we first locate the node in the trie, if it is an internal node, then $B_{24}$ is kept unchanged; otherwise, we assign $B_{24}[|p|] = (h > 0)$. Note that for one FIB update, we may need to update both the on-chip and off-chip lookup data structures. A router typically maintains the trie data structure on the control plane and uses it to compute the changes that need to be made to off-chip lookup data structures. Because little traffic hits the off-chip lookup data structures for levels 25∼32, updating the off-chip lookup data structures often can be performed in parallel with IP lookups on the on-chip data structures.

## IV. SAIL Optimization

In this section, we first present two optimization techniques of our SAIL algorithms, which favor the performance of FIB update and IP lookup, respectively. We use SAIL_U to denote SAIL_B with update oriented optimization, and SAIL_L to denote SAIL_B with lookup oriented optimization. Then, we extend SAIL_L to handle multiple FIBs.

### A. Update Oriented Optimization

*Data Structures & Lookup Process:* In this optimization, by prefix expansion, we push all solid nodes at levels $0 \sim 5$ to level 6, all solid nodes at levels $7 \sim 11$ to level 12, all solid nodes at levels $13 \sim 17$ to level 18, and all solid nodes at levels $19 \sim 23$ to level 24. With this 4-level pushing, looking up an IP address $\boldsymbol{a}$ is the same as without this pushing, except that if $B_{24}[\boldsymbol{a} \gg 8] = 0$, then we further check whether $B_{18}[\boldsymbol{a} \gg 14] = 1$, $B_{12}[\boldsymbol{a} \gg 20] = 1$, and $B_6[\boldsymbol{a} \gg 26] = 1$ till we get the first 1. This 4-level pushing brings two benefits to IP lookup. First, it reduces the maximum number of array lookups in on-chip memory from 24 to 4. Second, it reduces the on-chip memory usage by 49.2% because we do not need to store $B_0 \sim B_5$, $B_7 \sim B_{11}$, $B_{13} \sim B_{17}$, and $B_{19} \sim B_{23}$.

*FIB Update:* While improving lookup speed and reducing on-chip memory usage, this pushing incurs no extra cost to the update of on-chip data structures. With this pushing, we still achieve one on-chip memory access per FIB update because of three reasons. First, for any FIB update, it at most affects $2^6 = 64$ bits due to the above pushing. Second, typically by each memory access we can read/write 64 bits using a 64-bit processor. Third, as the lengths of the four s $B_6$, $B_{12}$, $B_{18}$, and $B_{24}$ are dividable by 64, the 64 bits that any FIB update needs to modify align well with word boundaries in on-chip memory. We implement each of these four s as an array of 64-bit unsigned integers; thus, for any FIB update, we only need to modify one such integer in one memory access.

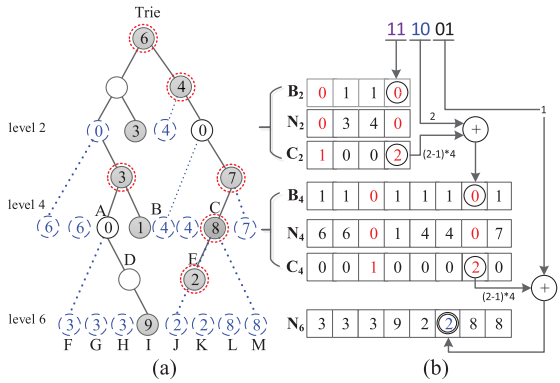


Fig. 3. Example data structure of SAIL_L. (a) is a trie. (b) includes bitmaps and next hop arrays.

### B. Lookup Oriented Optimization

*Data Structures:* In SAIL_B and SAIL_U, the maximum numbers of on-chip memory accesses are 24 and 4, respectively. To further improve lookup speed, we need to push nodes to fewer number of levels. On one hand, the fewer number of levels means the fewer numbers of on-chip memory accesses, which means faster lookup. On the other hand, pushing levels $0 \sim 23$ to 24 incurs too large on-chip memory. To trade-off between the number of on-chip memory accesses and the data structure size at each level, we choose two levels: one is between 0∼23, and the other one is 24. In our experiments, the two levels are 16 and 24. In this optimization, by prefix expansion, we first push all solid nodes at levels $0 \sim 15$ to level 16; second, we push all internal nodes at level 16 and all solid nodes at levels 17∼23 to level 24; third, we push all internal nodes at level 24 and all solid nodes at levels $25 \sim 31$ to level 32. We call this *3-level pushing*. For level 16, our data structure has three arrays: array $B_{16}[0..2^{16}-1]$, next hop array $N_{16}[0..2^{16}-1]$, and chunk ID array $C_{16}[0..2^{16}-1]$, where the chunk ID starts from 1. For level 24, our data structure has three arrays: array $B_{24}$, next hop array $N_{24}$, and chunk ID array $C_{24}$, where the size of each array is the number of internal nodes at level 16 times $2^8$. For level 32, our data structure has one array: next hop array $N_{32}$, whose size is the number of internal nodes at level 24 times $2^8$.

*Lookup Process:* Given an IP address $\boldsymbol{a}$, using $\boldsymbol{a}_{(i,j)}$ to denote the integer value of the bit string of $\boldsymbol{a}$ from the $i$-th bit to the $j$-th bit, we first check whether $B_{16}[\boldsymbol{a}_{(0,15)}] = 1$; if yes, then the $\boldsymbol{a}_{(0,15)}$-th node at level 16 is a solid node and thus the next hop for $\boldsymbol{a}$ is $N_{16}[\boldsymbol{a}_{(0,15)}]$; otherwise, then the $\boldsymbol{a}_{(0,15)}$-th node at level 16 is an empty node and thus we need to continue the search at level 24, where the index is computed as $(C_{16}[\boldsymbol{a}_{(0,15)}] - 1) * 2^8 + \boldsymbol{a}_{(16,23)}$. At level 24, denoting $(C_{16}[\boldsymbol{a}_{(0,15)}] - 1) * 2^8 + \boldsymbol{a}_{(16,23)}$ by $i$, the search process is similar to level 16: we first check whether $B_{24}[i] = 1$, if yes, then the $i$-th node at level 24 is a solid node and thus the next hop for $\boldsymbol{a}$ is $N_{24}[i]$; otherwise, the $i$-th node at level 24 is an empty node and thus the next hop must be at level 32, to be more precise, the next hop is $(C_{24}[i]-1)*2^8+\boldsymbol{a}_{(24,31)}$. Figure 3 illustrates the above data structures and IP lookup process where the three pushed levels are 2, 4, and 6. The pseudocode for the lookup process of SAIL_L is in Algorithm 2, where we use the bitmaps, next hop arrays, and the chunk ID arrays as separate arrays for generality and simplicity.

**Algorithm 2 SAIL_L**

**Input**: arrays: $B_{16}, B_{24}$
**Input**: Next hop arrays: $N_{16}, N_{24}, N_{32}$
**Input**: Chunk ID arrays: $C_{16}, C_{24}$
**Input**: $\boldsymbol{a}$: an IP address
**Output**: the next hop of the longest matched prefix.

```
if B16[a ≫ 16] = 1 then
    return N16[a ≫ 16]
end
i ← ((C16[a ≫ 16] − 1) ≪ 8) + (a ≪ 16 ≫ 24)
else if B24[i] = 1 then
    return N24[i]
end
else
    return N32[((C24[i] − 1) ≪ 8) + (a&255)]
end
```

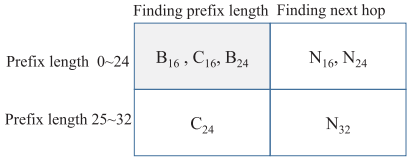


Fig. 4. Memory management for SAIL_L.

*Two-Dimensional Splitting:* The key difference between SAIL_L and prior IP lookup algorithms lies in its two-dimensional splitting. According to our two-dimensional splitting methodology, we should store the three arrays $B_{16}$, $C_{16}$, and $B_{24}$ in on-chip memory and the other four arrays $N_{16}$, $N_{24}$, $C_{24}$, and $N_{32}$ in off-chip memory as shown in Figure 4. We observe that for $0 \leqslant i \leqslant 2^{16}-1$, $B_{16}[i] = 0$ if and only if $N_{16}[i] = 0$, which holds if and only if $C_{16}[i] \neq 0$. Thus, the three arrays of $B_{16}$, $C_{16}$, and $N_{16}$ can be combined into one array denoted by $BCN$, where for $0 \leqslant i \leqslant 2^{16}-1$, $BCN[i]_{(0,0)} = 1$ indicates that $BCN[i]_{(1,15)} = N_{16}[i]$ and $BCN[i]_{(0,0)} = 0$ indicates that $BCN[i]_{(1,15)} = C_{16}[i]$. Although in theory for $0 \leqslant i \leqslant 2^{16}-1$, $C_{16}[i]$ needs 16 bits, practically, based on measurement from our real FIBs of backbone routers, 15 bits are enough for $C_{16}[i]$ and 8 bits for next hop; thus, $BCN[i]$ will be 16 bits exactly. For FPGA/ASIC platforms, we store $BCN$ and $B_{24}$ in on-chip memory and others in off-chip memory. For CPU/GPU/many-core platforms, because most lookups access both $B_{24}$ and $N_{24}$, we do combine $B_{24}$ and $N_{24}$ to $BN_{24}$ so as to improve caching behavior. $BN_{24}[i] = 0$ indicates that $B_{24}[i] = 0$, and we need to find the next hop in level 32; $BN_{24}[i] > 0$ indicates that the next hop is $BN_{24}[i] = N_{24}[i]$.

*FIB Update:* Given a FIB update of inserting/deleting/modifying a prefix, we first modify the trie that the router maintains on the control plane to make the trie equivalent to the updated FIB. Note that this trie is the one after the above 3-level pushing. Further note that FIB update messages are sent/received on the control plane where the pushed trie is maintained. Second, we perform the above 3-level pushing on the updated trie for the nodes affected by the update. Third, we modify the lookup data structures on the data plane to reflect the change of the pushed trie.

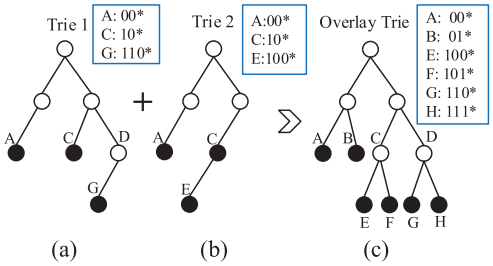


Fig. 5. Example of SAIL for multiple FIBs. (a) is the first tire. (b) is the second trie. (c) is the merged trie.

SAIL_L can perform FIB updates efficiently because of two reasons, although one FIB update may affect many trie nodes in the worst case. First, prior studies have shown that most FIB updates require only updates on a leaf node [60]. Second, the modification on the lookup arrays (namely the arrays, next hop arrays, and the chunk ID arrays) is mostly continuous, *i.e.*, a block of a lookup array is modified. We can use the *memcpy* function to efficiently write 64 bits in one memory access on a 64-bit processor.

### C. SAIL for Multiple FIBs

We now present our SAIL_M algorithm for handling multiple FIBs in virtual routers, which is an extension of SAIL_L. A router with virtual router capability (such as Cisco CRS-1/16) can be configured to run multiple routing instances where each instance has a FIB. If we build independent data structures for different FIBs, it will cost too much memory. One classic method is to merge all virtual FIBs into one, and then perform lookup on the merged one. Our design goal is to achieve constant on-chip memory and constant lookup time regardless of the number and the size of the virtual FIBs. No prior algorithm can achieve this goal. Next, we first show the classic merging method, and then show how to apply our SAIL framework to it.

*Classic Merging Method of Virtual FIBs:* Given $m$ FIBs $F_0, F_1, \cdots, F_{m-1}$, first, for each $F_i$ $(0 \leqslant i \leqslant m-1)$, for each prefix $p$ in $F_i$, we change its next hop $n_i(p)$ to a pair $(i, n_i(p))$. Let $F'_0, F'_1, \cdots$, and $F'_{m-1}$ denote the resulting FIBs. Second, we build a trie for $F'_0 \cup F'_1 \cup \cdots \cup F'_{m-1}$, the union of all FIBs. Note that in this trie, a solid node may have multiple (FIB ID, next hop) pairs. Third, we perform leaf pushing on this trie. Leaf pushing means to push each solid node to some leaf nodes [50]. After leaf pushing, every internal node is empty and has two children nodes; furthermore, each leaf node $v$ corresponding to a prefix $p$ is solid and has $m$ (FIB ID, next hop) pairs: $(0, n_0(p)), (1, n_1(p)), \cdots, (m-1, n_{m-1}(p))$, which can be represented as an array $\mathbb{N}$ where $\mathbb{N}[i] = n_i(p)$ for $0 \leqslant i \leqslant m-1$. Intuitively, we overlay all individual tries constructed from the $m$ FIBs, stretch all tries to have the same structure, and then perform leaf pushing on all tries.

Based on the overlay trie, we run the SAIL_L lookup algorithm. Note that in the resulting data structure, in each next hop array $N_{16}$, $N_{24}$, or $N_{32}$, each element is further an array of size $m$. Figure 5 shows two individual tries and the overlay trie.

*Lookup Process:* Regarding the IP lookup process for multiple FIBs, given an IP address $\boldsymbol{a}$ and a FIB ID $i$, we first use SAIL_L to find the next hop array $\mathbb{N}$ for $\boldsymbol{a}$. Then, the next hop for IP address $\boldsymbol{a}$ and a FIB ID $i$ is $\mathbb{N}[i]$.

*Two-Dimensional Splitting:* Regarding memory management, SAIL_M exactly follows the two-dimensional splitting

TABLE II
THEORETICAL BOUNDS FOR SAILS

| | On-chip memory | # on-chip mem acc per lookup | # on-chip mem acc per update |
|---|---|---|---|
| SAIL_B | = 4MB | 25 | 1 |
| SAIL_U | $\leqslant$2.03MB | 4 | 1 |
| SAIL_L | $\leqslant$2.13MB | 2 | Unbounded |
| SAIL_M | $\leqslant$2.13MB | 2 | Unbounded |

strategy illustrated in Figure 4. We store $BC_{16}$, which is the combination of $B_{16}$ and $C_{16}$, and $B_{24}$ in on-chip memory, and store the rest four arrays $N_{16}$, $N_{24}$, $C_{24}$, and $N_{32}$ in off-chip memory. The key feature of our scheme for dealing with multiple FIBs is that the total on-chip memory needed is bounded to $2^{16} * 17 + 2^{24} = 2.13$MB *regardless of the sizes, characteristics and number of FIBs.* The reason that we store $BC_{16}$ and $B_{24}$ in on-chip memory is that given an IP address $\boldsymbol{a}$, $BC_{16}$ and $B_{24}$ can tell us on which exact level, 16, 24, or 32 that we can find the longest matching prefix for $\boldsymbol{a}$. If it is at level 16 or 24, then we need 1 off-chip memory access as we only need to access $N_{16}$ or $N_{24}$. If it is at level 32, then we need 2 off-chip memory access as we need to access $C_{24}$ and $N_{32}$. Thus, the lookup process requires 2 on-chip memory accesses (which are on $BC_{16}$ and $B_{24}$) and at most 2 off-chip memory accesses.

*FIB Update:* Given a FIB update of inserting/deleting/ modifying a prefix, we first modify the overlay trie that the router maintains on the control plane to make the resulting overlay trie equivalent to the updated FIBs. Second, we modify the lookup data structures in the data plane to reflect the change of the overlay trie.

### D. Theoretical Bounds for SAILs

SAIL achieves contant yet small on-chip memory as shown in Table II. For SAIL_B, the upper bound of on-chip memory usage is $(\sum_{i=0}^{24} 2^i)/1024/1024/8 = 4\ MB$. For SAIL_U, the upper bound of on-chip memory usage is $(2^6+2^{12}+2^{18}+2^{24})/1024/1024/8 \approx 2.03\ MB$. For SAIL_L and SAIL_M, the upper bound of on-chip memory usage is $(2^{16}+2^{16}*16+2^{24})/1024/1024/8 \approx 2.13\ MB$ as we have a bitmap at level 16, and we also have a Chunk ID array at level 16. For on-chip memory accesses, the worst case of all SAIL algorithms ranges from 2 to 25. For off-chip memory accesses, the worst case of all SAIL algorithms is bounded to 2. For SAIL_M, the upper bounds for memory usage and the number of memory accesses are independent of the number of FIBs and the size of each FIB. For SAIL_B and SAIL_U, they only need 1 on-chip memory access per update. However, the update complexity of the other two algorithms is unbounded. In summary, as long as a routers has $\geqslant 4\ MB$ on-chip memory and the off-chip memory is large enough ($\geqslant 4\,GB$), the lookup speed of SAILs will be fast and bounded. No existing algorithms other than SAIL can achieve this.

### E. Comparison With Typical IP Lookup Algorithms

We qualitatively compare our algorithm SAIL_L with six well known algorithms in Table III in terms of time and space. Among all these algorithms, only our algorithm has the on-chip memory bound, supports four platforms, and supports virtual FIB lookup. Our algorithm is much faster than all the other algorithms. In the worst case, our algorithm only needs two off-chip memory accesses. All of these benefits are attributable to our two dimensional splitting and pivot pushing techniques. Our SAIL_L algorithm has the following two shortcomings. The first shortcoming of our SAIL_L algorithm is that the update speed in the worst case is unbounded. Fortunately, the average update speed is 2 off-chip memory access per update (see Figure 12). If one cares about the update performance most, we recommend the SAIL_U algorithm, which achieves O(1) on-chip update time, but is slower than SAIL_L. Our SAIL_L algorithm only supports CPU, GPU, FPGA and many-core platforms, and the second shortcoming is that it currently does not support multi-core platforms, which will be done in the future work.

## V. SAIL IMPLEMENTATION

### A. FPGA Implementation

We simulated SAIL_B, SAIL_U, and SAIL_L using Xilinx ISE 13.2 IDE. We did not simulate SAIL_M because its on-chip memory lookup process is similar to SAIL_L. In our FPGA simulation of each algorithm, to speed up IP lookup, we build one pipeline stage for the data structure corresponding to each level.

We use Xilinx Virtex 7 device (model XC7VX1140T) as the target platform. This device has 1,880 block RAMs where each block has 36 Kb, thus the total amount of on-chip memory is 8.26MB [1]. As this on-chip memory size is large enough, we implement the three algorithms of SAIL_B, SAIL_U, and SAIL_L on this platform.

### B. CPU Implementation

We implemented SAIL_L and SAIL_M on CPU platforms because their data structures have less number of levels as compared to SAIL_B and SAIL_U, and thus are suitable for CPU platform. For our algorithms on CPU platforms, the less number of levels means the less CPU processing steps. In contrast, in FPGA platforms, because we build one pipeline stage per level, the numbers of levels in our algorithms do not have direct impact on IP lookup throughput.

Our CPU experiments were carried out on an Intel(R) Core(TM) i7-3520M. It has two cores with four threads, each core works at 2.9 GHz. It has a 64KB L1 code cache, a 64KB L1 data cache, a 512KB L2 cache, and a 4MB L3 cache. The DRAM size of our computer is 8GB. The actual CPU frequency that we observe in our programs is 2.82GHz.

### C. GPU Implementation

We implemented SAIL_L in GPU platforms based on NVDIA's CUDA architecture [40]. In our GPU implementation, we store all data structures in GPU memory. Executing tasks on a GPU consists of three steps: (1) copy data from the CPU to the GPU, (2) execute multiple threads for the task, (3) copy the computing results back to the CPU; these three steps are denoted by *H2D, kernel execution, and D2H*, respectively. Thus, there is a natural communication bottleneck between the CPU and the GPU. To address this limitation, the common strategy is batch processing; that is, the CPU transfers a batch of independent requests to the GPU, and then the GPU processes them in parallel using many cores. In our GPU implementation, when packets arrive, the CPU first buffers them; when the buffer is full, the CPU transfers all their destination IP addresses to the GPU; when the GPU finishes all lookups, it transfers all lookup results back to the CPU. For our

TABLE III
COMPARISON WITH TYPICAL IP LOOKUP ALGORITHMS

| Algorithms | on-chip memory | average lookup speed | worst case lookup memory accesses | platform | virtual FIBs |
|---|---|---|---|---|---|
| SAIL_L | $\leqslant 2.13MB$ | around 700Mpps | 2 off-chip memory accesses | CPU, GPU, FPGA, many-core | support |
| PBF | – | – | 25 off-chip memory accesses | FPGA | – |
| Lulea | – | around 100Mpps | 12 off-chip memory accesses | CPU | – |
| D18R | – | around 150Mpps | 13 off-chip memory accesses | CPU | – |
| PopTrie$_{18}$ | – | around 430Mpps | 4 off-chip memory accesses | CPU | – |
| TreeBitmap | – | around 25Mpps | 6 off-chip memory accesses | FPGA | – |
| LC-trie | – | around 16Mpps | 9 off-chip memory accesses | CPU | – |

GPU based SAIL implementation, the data copied from the CPU to the GPU are IP addresses, and the computing results copied from the GPU to the CPU are the next hops for these IP addresses. In our GPU implementation, we employed two optimization techniques to speed up IP lookup performance: memory coalesce and multi-streaming.

The basic idea of memory coalesce is to let threads with continuous IDs read memory of continuous locations because GPU schedules its threads to access the main memory in the unit of 32 threads. In our GPU implementation, after the CPU transfers a batch of $32 * n$ IP addresses to the GPU's memory denoted as an array $A[0..32 * n - 1]$, if we assign threads $t_0, t_1, \cdots, t_{32*n-1}$ to process the IP addresses $A[0], A[1], \cdots, A[32 * n - 1]$, respectively, then all these $32 * n$ IP addresses can be read in $n$ memory transitions by the GPU, instead of $32 * n$ memory accesses.

The basic idea of multi-streaming, which is available on INVIDA Fermi GPU architecture, is to pipeline the three steps of H2D, kernel execution, and D2H. According to the CUDA computing model, data transfers (*i.e.*, H2D and D2H) and kernel executions within different streams can be parallelized via page-locked memory. A stream is a sequence of threads that must be executed in order. In our GPU implementation, we assign each $32 * k$ ($k \geqslant 1$) threads to a unique stream. Thus, different streams of $32 * k$ threads can be pipelined, *i.e.*, while one stream of $32 * k$ threads is transferring data from the CPU to the GPU, one stream of $32 * k$ threads is executing their kernel function, and another stream of $32 * k$ threads is transferring data from the GPU to the CPU.

### D. Many-Core Implementation

We implemented SAIL_L on the many-core platform Tilera TLR4-03680 [5], which has 36 cores and each core has a 256K L2 cache. Our experiments were carried out on a 64-bit operation system CentOS 5.9. One L2 cache access needs 9 cycles. In our many-core implementation, we let one core to serve as the main thread and all other cores to serve as lookup threads, where each lookup thread performs all steps for an IP address lookup.

### E. Deployment in PEARL

We deployed the SAIL_L algorithm in PEARL [53], which is a virtual router platform. It needs to perform IP lookup for each incoming packet. In default, it uses the path compressed trie [17], [49], which is a classic IP lookup algorithm. We replace the path compressed trie algorithm by our SAIL_L algorithm. Then we perform experiments to test the IP lookup speed using the real trace (see trace details in Section VI.A). Experimental results show that after deploying SAIL_L to PEARL, the lookup speed becomes around 28 times faster.

## VI. EXPERIMENTAL RESULTS

### A. Experimental Setup

To obtain real FIBs, we used a server to establish a peer relationship with a tier-1 router in China so that the server can receive FIB updates from the tier-1 router but does not announce new prefixes to the tier-1 router; thus, gradually, the server obtained the whole FIB from the tier-1 router. Note that it is hardly feasible to dump the FIB of a tier-1 router to hard disk, as the overhead incurred on the router is expensive and we are not permitted to do that due to the administrative policy. We use the *Quagga* to dump the FIB every hour [2]. We captured real traffic in one of the tier-1 routers interfaces for the first 10 minutes of each hour between October 22nd 08:00 AM 2013 to October 23rd 21:00 PM.

*Prefix Hit Analysis:* For the FIBs in most backbone routers, the traffic hits the prefixes of length $<= 24$. For the FIBs in AS border routers, there could be more traffic hitting prefixes of length $> 24$. As a result, the lookup speed of SAILs could be slower. However, the worst case of lookup speed is still 2 off-chip memory accesses.

In addition, the datasets of routing tables provided by RIPE NCC are archives of the snapshots. we downloaded 18 snapshots from www.ripe.net [3]. The time of first six snapshots collected from *rrc00* is 8:00 AM on January 1st of each year from 2008 to 2013, respectively, and are denoted by $FIB_{2008}, FIB_{2009}, \cdots, FIB_{2013}$. The rest twelve snapshots were taken at 08:00 AM on August 8 in 2013 from 12 different collectors, and thus are denoted by $rrc00, rrc01, \ rrc03, \cdots rrc07, \ rrc10, \cdots, rrc15$. We also generated 37 synthetic traffic traces. The first 25 traces contain packets with randomly chosen destinations. The other 12 traces were obtained by generating traffic evenly for each prefix in the 12 snapshots from the 12 collectors at 08:00 AM on August 8 in 2013. We call such traffic *prefix-based synthetic traffic*. The prefix based traffic is produced as follows: we generally guarantee that each prefix is matched with the same probability. In practice, however, when we generate traffic for a prefix, the produced traffic could match longer prefixes because of the LPM rule. We do not change the order of synthetic traffic. Specifically, given two prefixes $a_1$, $a_2$, suppose $a_2$ is after $a_1$ in the FIB, then the synthetic traffic for $a_2$ is also after those for $a_1$. As the source code for synthetic traffic is available at [4], the experimental comparison is fair.

As the website of ripe.net which is actually the largest open website for FIBs has FIBs from only these locations, we conducted experiments on all available FIBs, no cherry-pick. We have conducted experiments to test the percentage of longer prefixes, and the results are shown in Table IV. This table shows that the percentages of FIBs from different locations and different years are similar, around 1% to 2%.

We evaluated our algorithms on four metrics: *lookup speed* in terms of pps (# of packets per second), *on-chip memory size*

TABLE IV
THE PERCENTAGE OF LONG PREFIXES OF 23 IPV4 FIBS

| FIBs | Total Prefixes | Long Prefixes | Percentage |
|---|---|---|---|
| rrc00-2002 | 114069 | 2741 | 2.40% |
| rrc00-2003 | 121562 | 1709 | 1.41% |
| rrc00-2004 | 133214 | 1376 | 1.03% |
| rrc00-2005 | 165786 | 9768 | 5.89% |
| rrc00-2006 | 180150 | 1800 | 1.00% |
| rrc00-2007 | 210072 | 2564 | 1.22% |
| rrc00-2008 | 245393 | 4117 | 1.68% |
| rrc00-2009 | 291967 | 6293 | 2.16% |
| rrc00-2010 | 322152 | 6562 | 2.04% |
| rrc00-2011 | 348804 | 3780 | 1.08% |
| rrc00-2012 | 414300 | 5453 | 1.32% |
| rrc00-2013 | 495401 | 12178 | 2.46% |
| rrc01-2013 | 460380 | 1576 | 0.34% |
| rrc03-2013 | 476281 | 8492 | 1.78% |
| rrc04-2013 | 470772 | 5495 | 1.17% |
| rrc05-2013 | 467315 | 1179 | 0.25% |
| rrc06-2013 | 467915 | 1561 | 0.33% |
| rrc07-2013 | 465539 | 7034 | 1.51% |
| rrc10-2013 | 467723 | 6718 | 1.44% |
| rrc11-2013 | 472589 | 7357 | 1.56% |
| rrc12-2013 | 191601 | 2440 | 1.27% |
| rrc13-2013 | 373475 | 28216 | 7.55% |
| rrc14-2013 | 408900 | 4296 | 1.05% |
| rrc15-2013 | 371877 | 3094 | 0.83% |

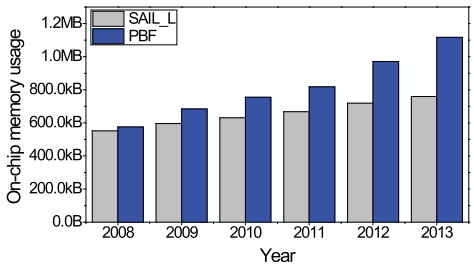


Fig. 6. On-chip memory usage over 6 years.

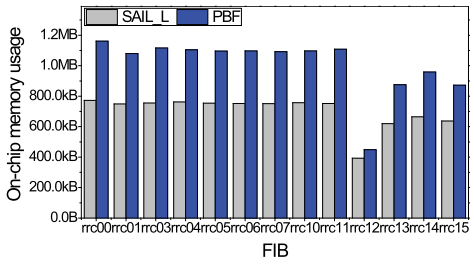


Fig. 7. On-chip memory usage of 12 FIBs.

in terms of MB, *lookup latency* in terms of microsecond, and *update speed* in terms of the total number of memory accesses per update. For on-chip memory sizes, we are only able to evaluate the FPGA implementation of SAIL algorithms. For lookup latency, we evaluated our GPU implementation because the batch processing incurs more latency.

We compared our algorithms with six well-known IP lookup algorithms: PBF [47], LC-trie [49], Tree Bitmap [52], Lulea Lulea [37], DXR [33], and Poptrie [7]. For DXR, we implemented its fastest version – D18R. We validated the correctness of all algorithms through exhaustive search: we first construct an exhaustive $2^{32} = 4G$ lookup table where the next hop of an IP address $\boldsymbol{a}$ is the $\boldsymbol{a}$-th entry in this table; second, for each IP address, we compare the lookup result with the result of this exhaustive table lookup, and all our implementations pass this validation.

### B. Performance on FPGA

We evaluated the performance of our algorithm on FPGA platforms in comparison with PBF, which is best suitable for FPGA platforms among the five well known algorithms, because the other five algorithms did not separate their data structures for on-chip and off-chip memory.

We first evaluate SAIL_L for on-chip memory consumption in comparison with PBF. Note that PBF stores its Bloom filters in on-chip memory. We compute the on-chip memory usage of PBF as follows. In [47], it says that PBF needs $1.003$ off-chip hash probes per lookup on average, given a routing table size of $116,819$. To achieve $1.003$ off-chip memory accesses per lookup assuming the best scenario of one memory access per hash probe, the overall false positive rate of the filters should be $0.003$. Thus, each Bloom filter should have a false positive rate of $0.003/(32-8)$ since PBF uses 24 filters. Assuming that these Bloom filters always achieve the optimal false positive, then from $0.003/(32-8) = (0.5)^k$, we obtain $k = 13$ and $m/n = 18.755$, where $m$ is the total size of all Bloom filters and $n$ is the number of elements stored in the filter. Thus, given a FIB with $n$ prefixes, the total on-chip memory usage of PBF is $18.755 * n$.

*Our experimental results on on-chip memory usage show that within the upper bound of 2.13MB, the on-chip memory usage of SAIL_L grows slowly and the growth rate is slower than PBF, and that the on-chip memory usage of SAIL_L is smaller than PBF*. For on-chip memory usage, the fundamental difference between SAIL_L and PBF is that the on-chip memory usage of SAIL_L has an upper bound but that of PBF grows with the number of prefixes in the FIB linearly without a practical upper bound. Figure 6 shows the evolution of the on-chip memory usage for both SAIL_L and PBF over the past 6 years based on our results on the 6 FIBs: $FIB_{2008}, FIB_{2009}, \cdots$, and $FIB_{2013}$. Figure 7 shows the on-chip memory usage of the 12 FIBs $rrc00, rrc01$, $rrc03, \cdots rrc07$, $rrc10, \cdots, rrc15$. Taking FIB $rrc00$ with 476,311 prefixes as an example, SAIL_L needs only 0.759MB on-chip memory.

We next evaluate SAIL_L for lookup speed on FPGA platform Xilinx Virtex 7. We did not compare with PBF because [47] does not provide implementation details for its FPGA implementation. We focus on measuring the lookup speed on the data structures stored in on-chip memory because off-chip memory lookups are out of the FPGA chip. As we implement the lookup at each level as one pipeline stage, SAIL_B, SAIL_U, SAIL_L have 24, 4, 2 pipeline stages, respectively. The more stages our algorithms have, the more complex of the FPGA logics are, and the slower the FPGA clock frequency will be. Our experimental results show that SAIL_B, SAIL_U, SAIL_L have clock frequencies of 351MHz, 405MHz, and 479MHz, respectively. As each of our pipeline stage requires only one clock cycle, *the lookup speed of SAIL_B, SAIL_U, SAIL_L are 351Mpps, 405Mpps, and 479Mpps, respectively*.

Let us have a deeper comparison of SAIL_L with PBF. The PBF algorithm without pushing requires $25 * k$ hash computations and memory accesses in the worst case because it builds 25 Bloom filters, each of which needs $k$ hash functions. With pushing, PBF needs to build at least 1 Bloom filter because the minimum number of levels is 1 (by pushing all nodes to level 32), although which is impractical. Further we assume that PBF uses the Kirsch and Mitzenmacher's double hashing

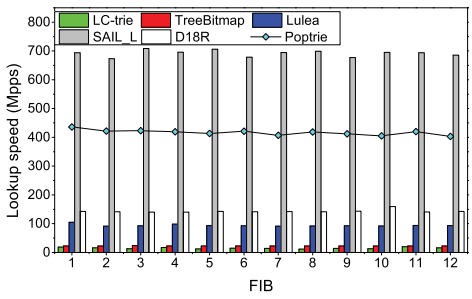


Fig. 8. Lookup speed with real traffic and real FIBs.

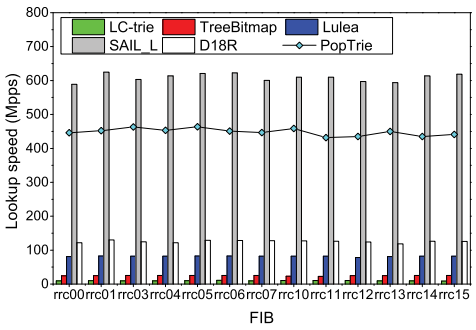


Fig. 9. Lookup speed with prefix-based traffic on 12 FIBs.

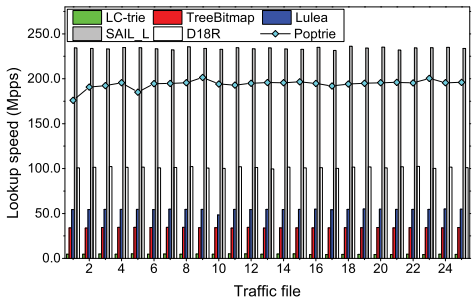


Fig. 10. Lookup speed with random traffic on $FIB_{2013}$.

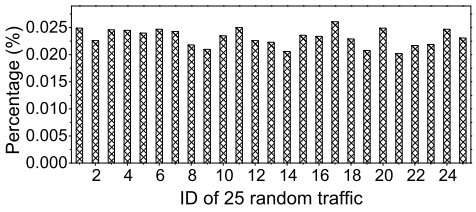


Fig. 11. Percentages of 25 traffic traces with more than 24 bits matching prefix length for the random traffic.

scheme based Bloom filters, which uses two hash functions to simulate multiple hash functions [6]. Although using the double hashing technique increases false positives, we assume it does not. Furthermore, suppose the input of hashing is 2 bytes, suppose PBF uses the well known CRC32 hash function, which requires 6.9 clock cycles per input byte. With these unrealistic assumptions, the number of cycles that PBF requires for searching on its on-chip data structures is $6.9 \times 2 \times 2$. In comparison, SAIL_L requires only 3∼10 instructions as discussed in Section V-B and needs only 4.08 cycles per lookup based on Figure 8. In summary, even with many unrealistic assumptions that favor PBF, SAIL_L still performs better.

### C. Performance on CPU

We evaluated the performance of our algorithm on CPU platforms in comparison with LC-trie, Tree Bitmap, Lulea, DXR's fastest version – D18R, and Poptrie. The CPU has two cores, but we only use one core to perform the lookups for all algorithms. We exclude PBF because it is not suitable for CPU implementation due to the many hashing operations.

*Our experimental results show that SAIL_L is several times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie*. For real traffic, SAIL_L achieves a lookup speed of 673.22∼708.71 Mpps, which is 34.53∼58.88, 29.56∼31.44, 6.62∼7.66, 4.36∼5.06, 1.59∼1.72 times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie, respectively. For prefix-based synthetic traffic, SAIL_L achieves a lookup speed of 589.08∼624.65 Mpps, which is 56.58∼68.46, 26.68∼23.79, 7.61∼7.27, 4.69∼5.04, 1.23∼1.32 times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie, respectively. For random traffic, SAIL_L achieves a lookup speed of 231.47∼236.08 Mpps, which is 46.22∼54.86, 6.73∼6.95, 4.24∼4.81, 2.27∼2.31, 1.13∼1.14 times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie, respectively. Figure 8 shows the lookup speed of these 6 algorithms with real traffic on real FIBs. The 12 FIBs are the 12 FIB instances of the same router during the first 12 hours period starting from October 22nd 08:00 AM 2013. For each FIB instance at a particular hour, the real traffic that we experimented is the 10 minutes of real traffic that we captured during that hour. Figure 9 shows the lookup speed of these 6 algorithms with prefix-based synthetic traffic on the 12 FIBs downloaded from www.ripe.net. Figure 10 shows the lookup speed of these 6 algorithms with random traffic on $FIB_{2013}$. From these figures, we further observe that for each of these 6 algorithms, its lookup speed on real traffic is faster than that on prefix-based traffic, which is further faster than that on random traffic. This is because real traffic has the best IP locality, which results in the best CPU caching behavior, and random traffic has the worst IP locality, which results in the worst CPU caching behavior.

As shown in Figure 11, the percentages of 25 traffic traces with more than 24 bits matching prefix length range from 0.021% to 0.025%. These percentages are much smaller than the ratio of longer prefixes of the FIB (2.3%), because longer prefixes cover smaller ranges. For example, a leaf prefix node at level 8 covers a range with a width of $2^{32-8}$, while a prefix node at level 32 only covers a range with a width of 1. All IP lookup algorithms perform much slower because the cache behaviour is very poor when randomly choosing an IP address from the 4GB space.

We now evaluate the FIB update performance of SAIL_L on the data plane. Figure 12 shows the variation of the number of memory accesses per update for 3 FIBs (rrc00, rrc01, and rrc03) during a period with $319 * 500$ updates. The average numbers of memory accesses per update for these three FIBs are 1.854, 2.253 and 1.807, respectively. The observed worst case is 7.88 memory accesses per update.

We now evaluate the lookup speed of SAIL_M for virtual routers. Figure 13 shows the lookup speed of SAIL_M algorithm as the number of FIBs grows, where $x$ FIBs means the first $x$ FIBs in the 12 FIBs $rrc00, rrc01, rrc03, \cdots rrc07$, $rrc10, \cdots, rrc15$, for both prefix-based traffic and random traffic. This figure shows that on average SAIL_M achieves 132 Mpps for random traffic and 366 Mpps for prefix-based traffic.

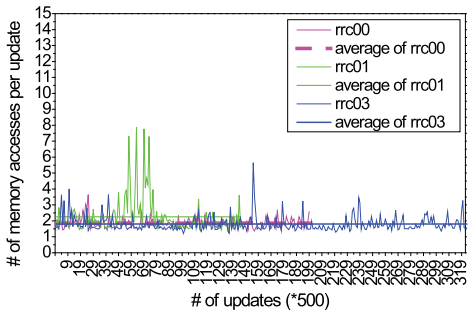


Fig. 12. # memory accesses per update.

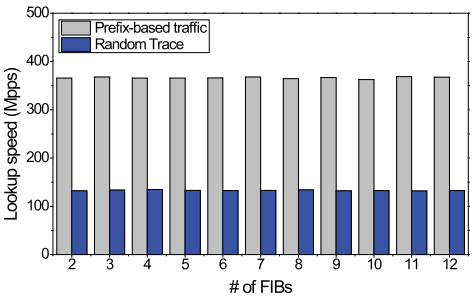


Fig. 13. Lookup speed of SAIL_M for 12 FIBs using prefix-based and random traffic.

*Lookup Speed vs. Percentage of Long Prefixes:* As shown in Figure 14, the lookup speed of SAIL_L degrades to 448∼467Mpps, 357∼380Mpps, 271∼296Mpps for having $1\%, 5\%, 10\%$ of longer prefixes, respectively. We also compared SAIL_L with Poptrie, which has lookup speeds of 162∼296Mpps, 150∼234Mpps, 124∼186Mpps for having $1\%, 5\%, 10\%$ of longer prefixes, respectively. In conclusion, SAIL_L is $1.47 \sim 2.83$ times faster than Poptrie. The performance degradation of both algorithms is mainly because all the longer prefixes are generated randomly, and the resulted lookup tables have more chunks to maintain comparing with the real FIBs that have better locality.

We exploit technological improvements that the authors of the other algorithms didn't have [55]. Basically, the amount of on-chip memory we have is vastly superior to what there was 10 years ago [54], [56], and already larger than the theoretical maximum value (4MB) that SAIL needs. So we can implement our lookup algorithm in a much more straightforward way than the others could. So we basically avoid some of the complexities they had to put in to fit whatever was available at the time.

The key idea of Poptrie [11] is similar to Lulea [36] as they both compress the next hop arrays based on bitmaps to achieve high memory efficiency. In Poptrie, it uses an SIMD (Single Instruction Multiple Data) instruction named Population Count (*i.e.*, *POPCNT*) which can count the number of 1s in a bitmap. As *POPCNT* is optimized by CPU internal logics, it is much faster than software solutions for counting 1s. Lulea was proposed in 1997 when CPUs did not support *POPCNT*. Thus, Poptrie achieves faster speed than Lulea. Poptrie has four weaknesses when compared with SAIL. First, Poptrie has no memory upper bound and much more memory accesses in the worst case. However, SAIL_L has an upper bound (*i.e.*, 2.13MB) for on-chip memory usage, and at most 2 off-chip memory accesses per lookup. Second, SAIL is much faster than Poptrie. Our experimental results on real and synthetic datasets in Section VI.C show that SAIL_L is $1.47 \sim 2.83$ times faster than Poptrie. Note that the Poptrie authors didn't use our open-sourced SAIL implementation [4], we believe their implementation probably introduces some unnecessary overhead that could potentially degrade the performance of SAIL algorithm. And We use their open source codes to conduct experimental comparison. Third, Poptrie was only implemented in CPU platform, and cannot be implemented on other platforms that do not support *POPCNT*, such as Many-core platform Tirela [5] and FPGA [1] platform. Fourth, the query speed of *POPCNT* will degrade linearly as the number of virtual FIBs increases.

Moreover, the Poptrie paper claims that SAIL cannot compile due to its structural limitation. In our previous paper, in level 16, we indeed use 1 bit as flag, and other 15 bits as next hop or ID of chunks. When the FIB is very large, 15 bits for chunk IDs could not be enough. This is a tiny problem that can be addressed very easily: we can use only 16 bits for next hop or chunk ID, and use additional $2^{16}$ bits as the flag.

### D. Evaluation on GPU

We evaluate SAIL_L on GPU platform with CUDA 5.0. We carry out these experiments on a DELL T620 server with an Intel CPU (Xeon E5-2630, 2.30 GHz, 6 Cores) and an NVIDIA GPU (Tesla C2075, 1147 MHz, 5376 MB device memory, 448 CUDA cores). These experiments use the 12 FIBs $rrc00, rrc01, rrc03, \cdots rrc07, rrc10, \cdots, rrc15$ and prefix-based traffic. We measure the lookup speed and latency with a range of CUDA configurations: the number of streams $(1, 2, \cdots, 24)$, the number of blocks per stream $(64, 96, \cdots, 512)$, and the number of threads per block $(128, 256, \cdots, 1024)$. The lookup speed is calculated as the total number IP lookup requests divided by the total lookup time. We use the Nvidia Visual Profiler tool to measure the lookup latency.

We evaluated the IP lookup speed versus traffic size (*i.e.*, the number of IP addresses in one batch of data sent from the CPU to the GPU). We generate 3 traffic traces of 3 different sizes 30K, 60K, 90K. Figure 15 shows our experimental results, from which we observe that larger traffic sizes lead to higher lookup speed. For the traffic size of 30K, SAIL_L achieves a lookup speed of 257∼322 Mpps. For the traffic size of 60K, SAIL_L achieves a lookup speed of 405∼447 Mpps. For the traffic size of 90K, SAIL_L achieves a lookup speed of 442∼547 Mpps.

We evaluated the IP lookup latency versus traffic size. Figure 16 shows our experimental results, from which we observe that larger traffic sizes lead to higher lookup latency. For the traffic size of 30K, SAIL_L has a lookup latency of 90∼124 $\mu s$. For the traffic size of 60K, SAIL_L has a lookup latency of 110∼152 $\mu s$. For the traffic size of 90K, SAIL_L has a lookup latency of 122∼185 $\mu s$.

### E. Evaluation on Many-Core Platform

We evaluated the lookup speed of SAIL_L versus the number of cores. We conducted our experiments on the many-core platform Tilera TLR4-03680. *Our experimental results show that the lookup rate increases linearly as the number of cores grows.* Note that we only have the results of 35 cores, because one core is responsible for traffic distribution and results collection. Figure 17 shows the results on FIB $rrc00$

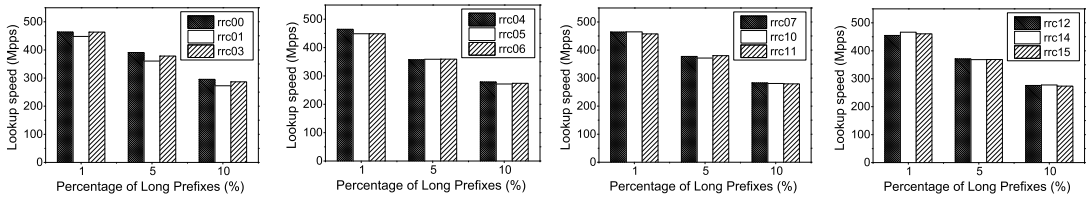


Fig. 14. Lookup speed with different percentage of long prefixes on 12 FIBs. (a) rrc: 00, 01, 03. (b) rrc: 04, 05, 06. (c) rrc: 07, 10, 11. (d) rrc: 12, 14, 15.

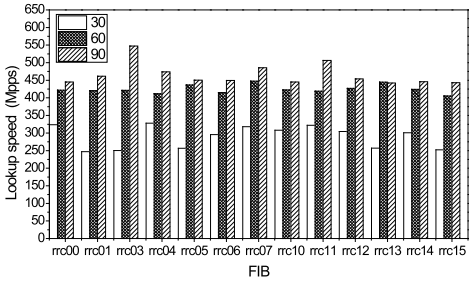


Fig. 15. Lookup speed VS. traffic size.

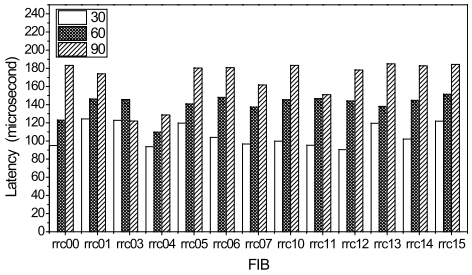


Fig. 16. Lookup latency VS. traffic size.

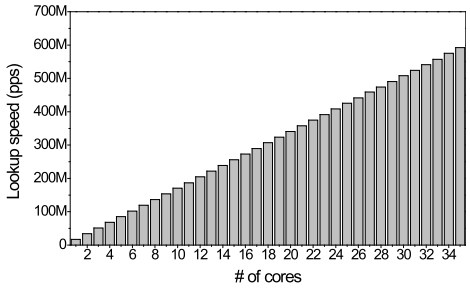


Fig. 17. Lookup speed VS. # of cores.

using prefix-based traffic. We have observed similar results for other FIBs.

## VII. Scalability for IPv6

### A. SAIL for Small IPv6 FIBs

Our SAIL framework is mainly proposed for IPv4 lookup; however, it can be extended for IPv6 lookup as well. An IPv6 address has 128 bits, the first 64 bits represent the network address and the rest 64 bits represent the host address. An IPv6 prefix has 64 bits. The real IPv6 FIBs in backbone routers from www.ripe.net only has around 14,000 entries, which are much smaller than IPv4 FIBs. To deal with IPv6 FIBs, we can push trie nodes to 6 levels of 16, 24, 32, 40, 48, and 64. To split along the dimension of prefix lengths, we perform the splitting at level 48. In other words, we store the and chunk ID arrays of levels 16, 24, 32, 40, and the array

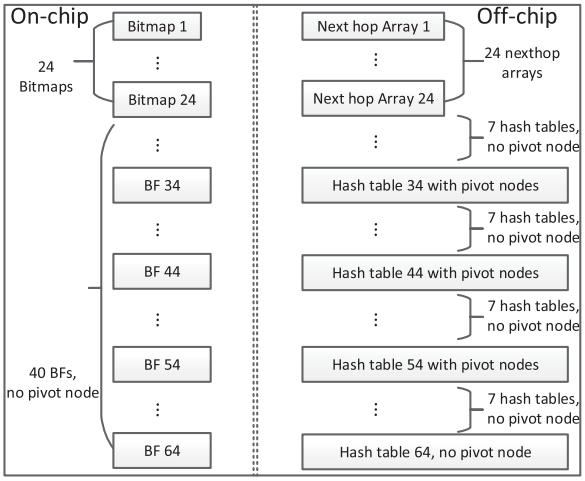


Fig. 18. A hybrid architecture for large IPv6 FIBs.

of level 48 in on-chip memory. Our experimental results show that the on-chip memory for an IPv6 FIB is about 2.2MB. Although the on-chip memory usage for IPv6 is much larger than that for IPv4 in the worst case because IPv6 prefix length are much longer than IPv4, as IPv6 FIB sizes are orders of magnitude smaller than IPv4 FIB sizes, the one-chip memory usage for IPv6 is similar to that for IPv4.

### B. SAIL for Large IPv6 FIBs

For large IPv6 FIBs, since the IPv6 address is much longer than IPv4 address, we can combine the methods of bitmaps in SAIL_U and Bloom filters. Specifically, we can split the prefix length into several parts. Here we give an example (see Figure 18): length 0∼24, length 25∼34, length 35∼44, length 45∼54, and length 55∼64. We propose a *hybrid* scheme that uses bitmaps for the prefixes with length 0∼24 and Bloom filters (BF) for others. Bloom filters have false positives whereas bitmaps do not have. The worst case of using Bloom filter (PBF [47]) is when all Bloom filters report true. Although this worst case happens with an extremely small probability, we need to check all the hash tables at level $25 \sim 64$, which is time-consuming.

*Pivot Inheriting:* To reduce the overhead in the worst case, we propose a novel scheme named *Pivot Inheriting*. Pivot inheriting is similar to our pivot pushing scheme. First, similar to pivot pushing, we choose several pivot levels. Second, for each pivot level, we only focus on the *Internal and Empty nodes (IE nodes)*. For each IE node, we set its next hop to the next hop of its nearest ancestor solid node. After the IE nodes inherit next hops, we call them *pivot nodes*. Then we show how pivot inheriting alleviates the worst case. Basically, after using pivot inheriting, we insert those pivot nodes in the

TABLE V
ON-CHIP MEMORY USAGE

| | rrc00 | rrc01 | rrc03 | rrc04 | rrc05 | rrc06 | rrc07 | rrc10 | rrc11 | rrc12 | rrc13 | rrc14 | rrc15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #prefixes (K) | 455.9 | 422.6 | 437.9 | 432.5 | 429.5 | 430 | 428 | 429.6 | 434.6 | 175.8 | 342.7 | 375.7 | 340.7 |
| BF memory (MB) | 1.04 | 0.97 | 1.00 | 0.99 | 0.98 | 0.98 | 0.98 | 0.98 | 0.99 | 0.40 | 0.78 | 0.86 | 0.78 |
| total memory (MB) | 5.04 | 4.97 | 5.00 | 4.99 | 4.98 | 4.98 | 4.98 | 4.98 | 4.99 | 4.40 | 4.78 | 4.86 | 4.78 |
| free memory (%) | 24% | 25% | 24% | 24% | 24% | 24% | 25% | 24% | 24% | 33% | 28% | 26% | 28% |

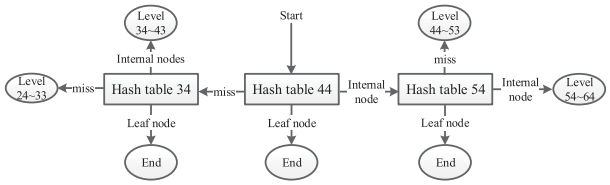


Fig. 19. Lookup process in the worst case.

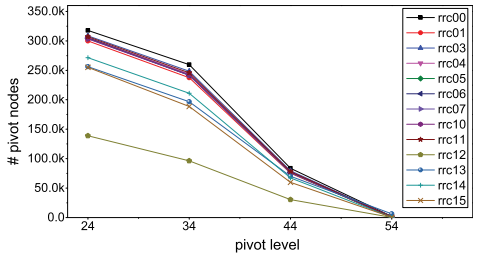


Fig. 20. The number of pivot nodes at the pivot levels.

corresponding hash table, but do not insert them in the Bloom filters.

*Example:* For example, as shown in Figure 18, we select these pivot levels: 24, 34, 44, and 54, and then carry pivot inheriting at these pivot levels. Then, we insert pivot nodes (their prefixes and next hops) into the corresponding hash tables at the pivot levels. Given an incoming IP address ***a***, when ***a*** matches an internal node at level 24 and all Bloom filters report true, we have the worst case. In this case, our lookup scheme proceeds in the following steps (see Figure 19):

Step I: We search ***a*** in the hash table at level 44: if the hash table reports a miss, then go to step II; if the hash table reports a leaf node, then the algorithm ends; if the hash table reports an internal node, then go to step III;

Step II: We search ***a*** in the hash table at level 34: if the hash table reports a miss, then we know that the matched level is in 24∼33; if the hash table reports a leaf node, then the algorithm ends; if the hash table reports an internal node, then we know that the matched level is in 34∼43;

Step III: We search ***a*** in the hash table at level 54: if the hash table reports a miss, then we know the matched level is in 44∼53; if the hash table reports a leaf node, then the algorithm ends; if the hash table reports an internal node, then we know the matched level is in 54∼64.

Thus, the worst case of the number of hash table searches can be reduced from 40 to 12. More pivot levels can further reduce the number of hash table searches at the cost of more off-chip memory usage. The overhead of our method is reasonably more off-chip memory usage, and we will show it in the experiments results. We need a flag with 2 bits for each node to indicate whether the node is a leaf node, or an internal node, or a pivot node.

### C. Experimental Results for Large IPv6 FIBs

*1) Experimental Setup:* The real-world IPv6 FIBs are small and only have around 13K entries. To evaluate the performance of our algorithm for large IPv6 FIBs, we synthesize IPv6 FIBs based on the real IPv4 FIBs downloaded from www.ripe.net [3] using the synthetic method proposed in [51]. The sizes of synthetic large IPv6 FIBs are similar to those of IPv4 FIBs. The IPv6 FIB sizes are around 10K, while the IPv4 FIB sizes are around 660K in 2017, 66 times larger. The IPv4 FIBs increase around 15% every year. Suppose IPv6 FIBs also increase 15% every year. Then it will take 30 years for IPv6 FIBs to reach the current size of IPv4 FIBs. We conduct experiments for large IPv6 FIBs (see Table IV), and results show that our algorithm works well. This means that our algorithm will work well for next 30 years without upgrading hardware.

*2) On-Chip Memory Usage:* The on-chip memory usage of our scheme includes two parts: 24 bitmaps and 40 BFs. The maximum memory usage of 24 bitmaps is 4MB. We evaluate the memory usage of 40 BFs using the same parameters as the experiments on PBF in Section VI-B: $k = 13$ and $m/n = 18.755$. The experimental results are shown in Table V, when using synthetic large IPv6 tables. We generate one IPv6 table according to one IPv4 FIB. The sizes of synthetic IPv6 FIBs are similar to the corresponding IPv4 FIBs. For example, the size of IPv4 table of rrc00 is 455 900, and the size of the synthetic IPv6 table is also 455 900. The number of prefixes at level 25∼64 is around 400K, and the BF memory usage is around 1MB. Thus, the total memory is around 5MB, while the on-chip memory capacity of old-fashioned FPGA is 6.6MB [1]. Modern ASICs have 50∼100MB on-chip memory [35].

*3) Additional Off-Chip Memory Usage:* With pivot inheriting, the worst case of this hybrid scheme can be reduced to 12 hash tables searches when using 4 pivot levels. The overhead of pivot inheriting is the additional off-chip memory usage.

We carry out pivot inheriting using the large synthetic IPv6 FIBs based on the IPv4 FIBs downloaded from [3]. We choose the four levels (24, 34, 44, and 54) as the pivot levels. In practice, for IPv4 FIBs, users can flexibly choose appropriate pivot levels. Here we only show an example for synthetic FIBs. It can be observed from Figure 20 that the number of pivot nodes is around 300K at level 24, but only 1K at level 54. The total number of pivot nodes is a little large than the size of synthetic IPv6 FIBs. Fortunately, these pivot nodes only consume additional off-chip memory, but do not consume on-chip memory at all. We show that the additional memory usage incurred by pivot inheriting in Figure 21.

Figure 21 shows the additional and total memory usage using pivot inheriting. It also shows the memory usage of Tree Bitmap on large synthetic FIBs. The source code of Tree Bitmap is available at [4]. For the sake of simplicity, we assume that all off-chip data structures use hash tables. To achieve small probability of hash collisions, we set the load

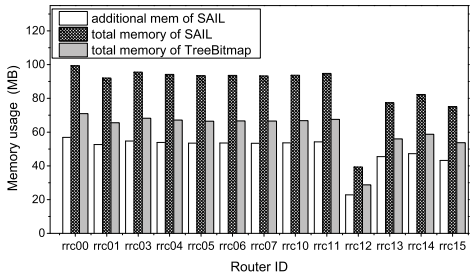


Fig. 21. The additional and overall off-chip memory usage.

factor to be 0.1. This means that for $n$ elements, we use $10 \times n$ hash buckets. Using these parameter settings, the experimental results are shown in Figure 21. It can be seen that the additional memory usage ranges from 22.8 MB to 56.9 MB with a mean of 49.6 MB, while the total memory usage ranges from 39.3 MB to 99.3 MB with a mean of 86.5 MB. Nowadays, the off-chip memory capacity is about 100 times of the above total memory usage of the hash tables. Since the off-chip memory is cheap, the overhead of pivot inheriting is reasonable and affordable. Experimental results also show that the total memory usage of Tree Bitmap ranges from 28.7 MB to 71.1 MB with a mean of 61.8, and it is about 71.4 % of that of our SAIL algorithm. However, Tree Bitmap does not split the memory into on-chip and off-chip memory, and one lookup needs multiple off-chip memory accesses. In contrast, we only need less than 5 MB on-chip memory usage, and $10\times$ buckets in the off-chip hash tables to achieve the speed of about 1 off-chip memory access per lookup.

*Analysis of IPv6 Lookup Performance:* In the best case, all Bloom filters have no error, then the lookup only needs one hash table probe, which can be considered as one memory access [47]. In the worst case, the longest matched prefix is at level 24 or 54, and all Bloom filters report yes. Note that Bitmaps have no false positive. In this case, our algorithm needs $2 + 10$ hash table probes, *i.e.*, 12 off-chip memory accesses. As we use 13 hash functions for each Bloom filter, the false positive rate is $0.5^{13} = 2^{-13}$ when using the optimal setting. The probability that the worst case happens is $0.5^{13*(10-1)} = 2^{-117}$. On average, each lookup needs around one hash table probe.

## VIII. Conclusion

We make four key contributions in this paper. First, we propose a two-dimensional splitting approach to IP lookup. The key benefit of such splitting is that we can solve the sub-problem of finding the prefix length $\leqslant 24$ in on-chip memory of bounded small size, with the help of our proposed *pivot pushing* algorithm. Second, we propose a suite of algorithms for IP lookup based on our SAIL framework. One key feature of our algorithms is that we achieve constant, yet small, IP lookup time and on-chip memory usage. Another key feature is that our algorithms are cross platform as the data structures are all arrays and only require four operations of ADD, SUBTRACTION, SHIFT, and logical AND. Note that SAIL is a general framework where different solutions to the sub-problems can be adopted. The algorithms proposed in this paper represent particular instantiations of our SAIL framework. Third, we extend our SAIL approach to IPv6, and proposed a hybrid scheme. To improve the performance of synthetic large IPv6 FIBs in the worst case, we propose a novel technique called *pivot inheriting*. Fourth, we implemented our algorithms on four platforms (namely FPGA, CPU, GPU, and many-core) and conducted extensive experiments to evaluate our algorithms using real FIBs and traffic from a major ISP in China. Our experimental results show that our SAIL algorithms are several times or even two orders of magnitude faster than the well known IP lookup algorithms. Furthermore, we have open sourced our SAIL_L algorithm and four well known IP lookup algorithms (namely LC-trie, Tree Bitmap, Lulea, and DXR) that we implemented in [4].

## References

[1] *FPGA Data Sheet*. Accessed: Aug. 2, 2015. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[2] *Quagga Routing Suite*. Accessed: Oct. 10, 2013. [Online]. Available: http://www.nongnu.org/quagga/

[3] *RIPE Network Coordination Centre*. Accessed: Dec. 7, 2013. [Online]. Available: https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data

[4] *SAIL Webpage*. Accessed: Jun. 8, 2017. [Online]. Available: http://net.pku.edu.cn/~yangtong/pages/SAIL.html

[5] *Tilera Datasheet*. Accessed: Oct. 10, 2013. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02_web.pdf

[6] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter," in *Proc. Eur. Symp. Algorithms*. Berlin, Germany: Springer, 2006, pp. 456–467.

[7] 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, pp. 57–70.

[8] H. Dai, L. Meng, and A. X. Liu, "Finding persistent items in distributed, datasets," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1–9.

[9] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," *Proc. VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[10] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy Bloom filters for multi-set membership testing," in *Proc. ACM SIGMETRICS*, 2016, pp. 139–151.

[11] D. Pao, Z. Lu, and Y. H. Poon, "IP address lookup using bit-shuffled trie," *Comput. Commun.*, vol. 47, pp. 51–64, Jul. 2014.

[12] D. Shah and P. Gupta, "Fast incremental updates on Ternary-CAMs for routing lookups and packet classification," in *Proc. Hot Interconnects*, 2000, pp. 1–9.

[13] F. Wang and M. Hamdi, "Matching the speed gap between SRAM and DRAM," in *Proc. IEEE Int. Conf. High Perform. Switching Routing*, May 2008, pp. 104–109.

[14] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Proc. IEEE ISCA*, Jun. 2005, pp. 123–133.

[15] F. Zane, G. Narlikar, and A. Basu, "Coolcams: Power-efficient TCAMs for forwarding engines," in *Proc. IEEE INFOCOM*, Mar./Apr. 2003, pp. 42–52.

[16] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," in *Proc. ACM SIGCOMM*, 2013, pp. 111–122.

[17] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*, vol. 2. Reading, MA, USA: Addison-Wesley, 1991.

[18] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM*, Mar./Apr. 1998, pp. 1240–1247.

[19] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for IP address lookup," in *Proc. IEEE ANCS*, Oct. 2005, pp. 81–90.

[20] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Scalable IP lookups using shape graphs," in *Proc. IEEE Int. Conf. Netw. Protocols*, Oct. 2009, pp. 73–82.

[21] H. Le, W. Jiang, and V. K. Prasanna, "A SRAM-based architecture for Trie-based IP lookup using FPGA," in *Proc. IEEE FCCM*, Apr. 2008, pp. 33–42.

[22] K. Huang, G. Xie, Y. Li, and A. X. Liu, "Offset addressing approach to memory-efficient IP address lookup," in *Proc. IEEE INFOCOM*, 2011, pp. 306–310.

[23] H. Lim, C. Yim, E. E. Swartzlander, "Priority tries for IP address lookup," *IEEE Trans. Comput.*, vol. 59, no. 6, pp. 784–794, Jun. 2010.

[24] J. Fu and J. Rexford, "Efficient IP-address lookup with a shared forwarding table for multiple virtual routers," in *Proc. ACM CoNEXT*, 2008, Art. no 21.

[25] K. Zheng, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 863–875, Aug. 2006.

[26] K. Sklower, "A tree-based packet routing table for berkeley unix," in *Proc. USENIX Winter*, 1991, pp. 93–99.

[27] L. Luo *et al.*, "A trie merging approach with incremental updates for virtual routers," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1222–1230.

[28] H. Le and V. K. Prasanna, "Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1026–1039, Jul. 2012.

[29] H. Lim and N. Lee, "Survey and proposal on binary search algorithms for longest prefix match," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 3, pp. 681–697, 3rd Quart., 2012.

[30] H. Lim, K. Lim, N. Lee, and K.-H. Parl, "On adding Bloom filters to longest prefix matching algorithms," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 411–423, Feb. 2014.

[31] M. Meribout and M. Motomura, "A new hardware algorithm for fast IP routing targeting programmable routers," in *Proc. Int. Conf. Network Control Eng. QoS, Secur. Mobility*. Boston, MA, USA: Springer, 2003, pp. 164–179.

[32] W. Marcel, V. George, T. Jon, and P. Bernhard, "Scalable high speed IP routing lookups," in *Proc. ACM SIGCOMM*, 1997, pp. 25–36.

[33] M. Zec, L. Rizzo, and M. Mikuc, "DXR: Towards a billion routing lookups per second in software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 29–36, Sep. 2012.

[34] M. Bando and H. J. Chao, "Flashtrie: Hash-based prefix-compressed trie for IP route lookup beyond 100Gbps," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 821–829.

[35] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 15–28.

[36] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Netw.*, vol. 15, no. 2, pp. 8–23, Mar./Apr. 2001.

[37] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, 1997, pp. 3–14.

[38] M. Mittal, "Deterministic lookup using hashed key in a multi-stride compressed trie structure," U.S. Patent 7 827 218 B1, Oct. 29, 2013.

[39] M. J. Akhbarizadeh, M. Nourani, R. Panigrahy, and S. Sharma, "A TCAM-based parallel architecture for high-speed packet forwarding," *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 58–72, Jan. 2007.

[40] *NVIDIA CUDA C Best Practices Guide, Version 5.0*, NVIDIA Corp., Santa Clara, CA, USA, Oct. 2012.

[41] P. Crescenzi, L. Dardini, and R. Grossi, "IP address lookupmade fast and simple," in *Proc. Eur. Symp. Algorithms*. Berlin, Germany: Springer, 1999, pp. 65–76.

[42] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: Scalable IP lookup with fast updates," *Comput. Netw.*, vol. 44, no. 3, pp. 289–303, 2004.

[43] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed IP lookup algorithms," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 802–812, Aug. 2005.

[44] R. Panigrahy and S. Sharma, "Reducing tcam power consumption and increasing throughput," in *Proc. High Perform. Interconnects*, Aug. 2002, pp. 107–112.

[45] S. Sahni and H. Lu, "Dynamic tree bitmap for ip lookup and update," in *Proc. 6th Int. Conf. Netw. (ICN)*, Apr. 2007, p. 79.

[46] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. ACM SIGCOMM*, 2010, pp. 195–206.

[47] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," in *Proc. ACM SIGCOMM*, 2003, pp. 201–212.

[48] H. Song, M. Kodialam, F. Hao, and T. Lakshman, "Building scalable virtual routers with trie braiding," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[49] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1083–1092, Jun. 1999.

[50] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, 1999.

[51] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-random generator for IPv6 tables," in *Proc. IEEE HPI*, Aug. 2004, pp. 35–40.

[52] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software IP lookups with incremental updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[53] G. Xie *et al.*, "PEARL: A programmable virtual router platform," *IEEE Commun. Mag.*, vol. 49, no. 7, pp. 71–77, Jul. 2011.

[54] K. Xie *et al.*, "Fast tensor factorization for accurate Internet anomaly detection," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3794–3807, Dec. 2017.

[55] K. Xie *et al.*, "On-line anomaly detection with high accuracy," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1222–1235, Jun. 2018.

[56] K. Xie *et al.*, "Accurate recovery of Internet traffic data: A sequential tensor completion approach," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 793–806, Apr. 2018.

[57] T. Yang *et al.*, "Clue: Achieving fast update over compressed table for parallel lookup with reduced dynamic redundancy," in *Proc. IEEE ICDCS*, Jun. 2012, pp. 678–687.

[58] T. Yang *et al.*, "A shifting framework for set queries," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3116–3131, Oct. 2017.

[59] T. Yang *et al.*, "A shifting Bloom filter framework for set queries," *ACM J. VLDB Endowment*, vol. 9, no. 5, pp. 408–419, Jan. 2016.

[60] T. Yang *et al.*, "An ultra-fast universal incremental update algorithm for trie-based routing lookup," in *Proc. ACM/IEEE ICNP*, Oct./Nov. 2012, pp. 1–10.

[61] T. Yang *et al.*, "Guarantee IP lookup performance with FIB explosion," in *Proc. ACM SIGCOMM*, 2014, pp. 39–50.

[62] T. Yang *et al.*, "Approaching optimal compression with fast update for large scale routing tables," in *Proc. IEEE Int. Workshop Quality Service*, Jun. 2012, pp. 1–9.

[63] J. Zhao, X. Zhang, X. Wang, and X. Xue, "Achieving O(1) IP lookup on GPU-based software routers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 429–430, 2010.


**Tong Yang** received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences, China, from 2013 to 2014. He is currently a Research Assistant Professor with the Computer Science Department, Peking University. He published papers in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, ICDE, and INFOCOM. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches, and KV stores.

**Gaogang Xie** received the Ph.D. degree in computer science from Hunan University, Changsha, China, in 2002. He is currently a Professor and the Director of the Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include programmable virtual routers, future Internet architecture, and Internet measurement.

**Alex X. Liu** received the Ph.D. degree in computer science from The University of Texas at Austin in 2006. His research interests focus on networking and security. He received the IEEE & IFIP William C. Carter Award in 2004, a National Science Foundation CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received Best Paper Awards from ICNP-2012, SRDS-2012, and LISA-2010. He is an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING, an Associate Editor of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and an Area Editor of *Computer Communications*.

**Qiaobin Fu** received the B.Eng. degree from the Dalian University of Technology in 2011 and the M.S. degree from the University of Chinese Academy of Sciences in 2014. He is currently pursuing the Ph.D. degree with Boston University in 2014. His research interests focus on computer networking, XIA, and cloud computing.

**Yanbiao Li** was born in 1986. He is currently a Post-Doctoral Fellow with Hunan University and a Visiting Research Associate with the Institute of Computing Technology, Chinese Academy of Science. His research interests focus on the future Internet architecture and high performance packet processing.

**Xiaoming Li** is currently a Professor in computer science and technology and the Director of the Institute of Network Computing and Information Systems, Peking University, China.

**Laurent Mathy** photograph and biography not available at the time of publication.