Guarantee IP Lookup Performance with FIB Explosion

Tong Yang,Gaogang Xie*
ICT, CAS, China.
yangtongemail@gmail.com,
*xie@ict.ac.cn

YanBiao Li
Hunan University, China
lybmath_cs@hnu.edu.cn

Qiaobin Fu
ICT, CAS, China
fuqiaobin@ict.ac.cn

Alex X. Liu
Department of Computer Science and Engineering, Michigan State University
alexliu@cse.msu.edu

Qi Li
ICT, CAS, China
liqi19872006@gmail.com

Laurent Mathy
University of Liege, Belgium
laurent.mathy@ulg.ac.be

ABSTRACT

The Forwarding Information Base (FIB) of backbone routers has been rapidly growing 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 IP lookup on prefixes of length 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 several times or even two orders of magnitude faster than well known IP lookup algorithms.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Store and forward networks*

Keywords

IP Lookup; SAIL; Virtual Router Multi-FIB Lookup; LPM

1. INTRODUCTION

1.1 Background and Motivation

The growth of FIB sizes on backbone routers has been accelerating. According to the RIPE Network Coordination Centre, FIB sizes have become about half a million 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 [9], 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. Without satisfying these two requirements, router performance will degrade as FIB grows, and router hardware will have to be upgraded periodically.

1.2 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 [11,19] and FPGA based schemes [14,16], but their on-chip memory usage will grow quickly as FIB sizes grow. Some algorithms, such as full-expansion [31] and DIR-24-8 [13], 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.

1.3 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

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626297.

two entities, this splitting allows us to potentially pipeline the processes of finding the prefix length and the next hop.

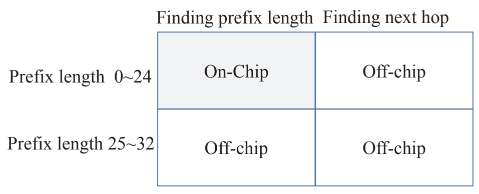

Figure 1: Two-dimensional splitting of IP lookup.

Second, we split the IP lookup problem into two sub-problems along the dimension of prefix length: length ≤ 24 and length ≥ 25. This splitting is based on our key observation that on backbone routers, for almost all traffic, the longest matching prefix has a length ≤ 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; however, the FIBs for edge routers are significantly smaller that those for backbone routers. 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 ≤ 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 [36]. Pierluigi *et al.* and Gupta *et al.* 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 [13, 31]. 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.

1.4 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 bit maps. Given a set of prefixes in a FIB, for each prefix length i ($0 \leq i \leq 32$), we first build a bit map $B_i[0..2^i-1]$ 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 bit maps 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 ≤ 24 and those with length ≥ 25. (1) For prefixes of length ≤ 24, for each prefix length i ($0 \leq i \leq 24$), 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 a, we first find its prefix length using bit maps in on-chip memory and find the next hop using one array lookup in off-chip memory. To find the prefix length using bit maps, for i from 24 to 0, we test whether $B_i[a >> (32-i)] = 1$; once we find the first i so that $B_i[a >> (32-i)] = 1$ holds, we know that the longest matching prefix length is i. Here $a >> (32-i)$ means right shifting 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 a is i, we can find its next hop $N_i[a >> (32-i)]$ by one off-chip memory access. (2) For prefixes of length ≥ 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 ≥ 25 depends on design priorities, but have little impact on the overall IP lookup performance because most traffic hits prefixes of length ≤ 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 bit maps; 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.

The remaining of the paper proceeds as follows. We first review related work in Section 2. In Section 3 and 4, we introduce our basic SAIL algorithm and optimization techniques, respectively. We then present our implementation details and experimental results in Section 5 and 6, respectively. We discuss the application of our SAIL framework to IPv6 lookup in Section 7. Finally, we give concluding remarks in Section 8.

2. 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 [27], path-compressed trie [20], k-stride multibit trie [39], full expansion/compression [31], LC-trie [38], Tree Bitmap [40], priority trie [17], Lulea [28], DIR-24-8 [13], flashtrie [26], shapeGraph [15], and trie-folding [12]. A comprehensive survey of trie-based algorithms is in [27].

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 [36]. Lim *et al.* proposed to use one bloom filter to find the longest matching prefix length [22]. 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.

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 [24], binary range search [27], multiway range trees [32], and range search using many cores [25].

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 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 [11]. Some schemes use multiple TCAM chips to improve lookup speed [19, 29, 34]. Devavrat *et al.* proposed to reduce the movement of prefixes for fast updating [8].

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 [23, 33]; 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 [14]. 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 [7, 10, 16].

GPU-based Algorithms: Leveraging the massive parallel processing capability of GPU, some schemes have been proposed to use GPU to accelerate IP lookup [35, 42].

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 [18, 21, 37].

3. SAIL BASICS

In this section, we present the basic version of our SAIL algorithms. In the next section, we will present some optimization techniques. Table 1 lists the symbols used throughout this paper.

3.1 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 the 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 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 bit map array $B_i[0..2^i-1]$ of length 2^i, and the initial values are all $0s$. At each level i $(0 \le i \le 32)$, 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 on level i. Figure 2(c) shows the bit maps for levels 0 to 4 of the trie in 2(b). Taking bit map 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 bit maps $B_0, B_1, \cdots, B_{32}$ that we constructed from the trie for a FIB, for any given IP address a, for i from 32 to 0, we test whether $B_i[a >> (32-i)]=1$; once we find the first i that $B_i[a >> (32-i)]=1$ holds, we know that the longest matching prefix length is i. Here $a >> (32-i)$ means right shifting a by $32-i$ bits. For each bit map 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 a, once we find its longest matching prefix length i, then we know its next hop is $N_i[a >> (32-i)]$.

Table 1: Symbols used in the paper

Symbol	Description
B_i	bit map 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
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
SAIL_B	SAIL basic algorithm
SAIL_U	SAIL_B + updated oriented optimization
SAIL_L	SAIL_B + lookup oriented optimization
SAIL_M	SAIL_L extension for multiple FIBs
$a_{(i,j)}$	integer value of bit string of a from i-th bit to j-th bit

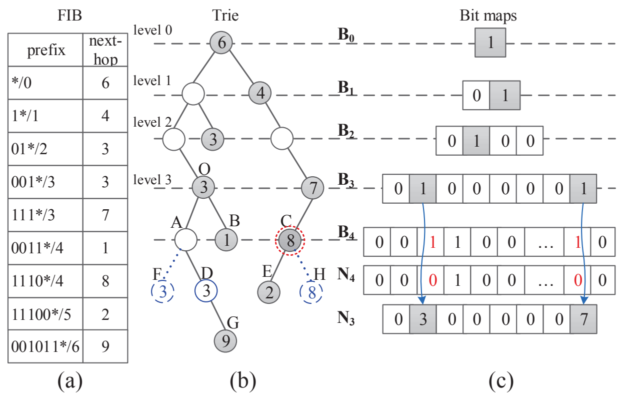

Figure 2: Basic SAIL Algorithm.

3.2 Splitting Prefix Length

Based on our observation that almost all the traffic of backbone routers has the longest matching prefix length ≤ 24, we split all prefixes in the given FIB into prefixes of length ≤ 24, which we call *short prefixes* and prefixes of length ≥ 25, which we call *long prefixes*. By this splitting, we want to store the bit maps of prefixes of length ≤ 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 a, traversing a from the root of the trie downward, for any internal or leaf node v that the traversal passes, we say *a passes v*. Based on the above concepts, we introduce Theorem 3.1.

THEOREM 3.1. *Given a trie constructed from a FIB, after pivot pushing, for any IP address a, 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 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 a's longest matching prefix is $p(v)$ (*i.e.*, the prefix represented by node v) and thus a's longest matching prefix is on the pivot level. For the second case where v is an internal node, because of pivot pushing, a must pass a solid node on a level below the pivot level, which means that a's longest matching prefix is below the pivot level. □

Based on Theorem 3.1, we construct the bit map for the pivot level l as follows: for any node v on level l, we assign $B_l[|p(v)|] = 1$; in other words, $B_l[i] = 0$ if and only if there is no node on level l that corresponds to the prefix denoted by i. Thus, given an IP address a, $B_l[a >> (32 - l)] = 1$ 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 $B_{24}[a >> 8] = 1$, we know whether the longest matching prefix length is ≤ 23 or ≥ 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 on 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 bit map array and a next hop array for each level of 0 to 24 as above. Note that for any i $(0 \leq i \leq 23)$ and any j $(0 \leq j \leq 2^i - 1)$, $B_i[j] = 1$ if and only if there is a solid node on level i that corresponds to the prefix denoted by j; for level 24 and any j $(0 \leq j \leq 2^{24} - 1)$, $B_{24}[j] = 1$ if and only if there is a node, no matter solid or empty, on 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 on 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 on level that corresponds to the prefix denoted by j, which means that the longest matching prefix length is ≥ 25. Thus, given an IP address a, if $B_{24}[a >> 8] = 0$, then we further check whether $B_{23}[a >> 9] = 1$, $B_{22}[a >> 10] = 1$, $\cdots$, $B_0[a >> 32] = 1$ until we find the first 1; if $B_{24}[a >> 8] = 1$, then we know a's longest matching prefix length is ≥ 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 $\sum_{i=0}^{24} 2^i = 4MB$. Consider the example in Figure 2. Given an address `001010`, as the pivot level is 4, since $B_4[001010 >> 2] = B_4[0010] = B_4[2] = 1$ and $N_4[001010 >> 2] = N_4[0010] = N_4[2] = 0$, then we know that the longest matching prefix length is longer than 4 and we will continue the search on levels below 4.

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

Algorithm 1: SAIL_B

```
Input: Bit map arrays: $B_0, B_1, \cdots, B_{24}$
Input: Next hop arrays: $N_0, N_1, \cdots, N_{24}$
Input: $a$: an IP address
Output: next hop of the longest matched prefix
1  if $B_{24}[a >> 8]=0$ then
2      for $j = 23; j > 0; j--$ do
3          if $B_j[a >> (32 - j)]=1$ then
4              return $N_j[a >> (32 - j)]$
5          end
6      end
7  end
8  else if $N_{24}[a >> 8]>0$ then
9      return $N_{24}[a >> 8]$;
10 end
11 else
12     lookup on levels 25∼32;
13 end
```

There are multiple ways to handle prefixes of length ≥ 25. Below we give one simple implementation using next hop arrays. Let the number of internal nodes on level 24 be n. We can push all solid nodes on levels 25∼31 to level 32. Afterwards, the number of nodes on level 32 is $256 * n$ because each internal node on 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 on levels from 25 to 31 to level 32, we do not need bit maps $B_{25}, B_{26}, \cdots, B_{32}$. Now consider the nodes on level 24. For each leaf node, its corresponding entry in bit map 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 bit map 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 a is simple: if $B_{24}[a >> 8] = 0$, then we know the longest matching prefix length is within $[0, 23]$ and further test whether $B_{23}[a >> 8] = 1$; if $B_{24}[a >> 8] = 1 \wedge N_{24}[a >> 8] > 0$, then we know that the longest matching prefix length is 24 and the next hop is $N_{24}[a >> 8]$; if $B_{24}[a >> 8] = 1 \wedge N_{24}[a >> 8] < 0$, then we know that the longest matching prefix length is longer than 24 and the next hop is $N_{32}[(|N_{24}[a >> 8]| - 1) * 256 + (a\&255)]$.

3.3 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 ≥ 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.

4. SAIL OPTIMIZATION

In this section, we first present two optimization techniques of our SAIL algorithms, which favors 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.

4.1 Update Oriented Optimization

Data Structures & Lookup Process: In this optimization, by prefix expansion, we push all solid nodes on levels $0 \sim 5$ to level 6, all solid nodes on levels $7 \sim 11$ to level 12, all solid nodes on levels $13 \sim 17$ to level 18, and all solid nodes on levels $19 \sim 23$ to level 24. With this 4-level pushing, looking up an IP address a is the same as without this pushing, except that if $B_{24}[a >> 8] = 0$, then we further check whether $B_{18}[a >> 14] = 1$, $B_{12}[a >> 20] = 1$, and $B_6[a >> 26] = 1$ 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 bit maps 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 bit maps 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.

4.2 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 on levels $0 \sim 15$ to level 16; second, we push all internal nodes on level 16 and all solid nodes on levels 17∼23 to level 24; third, we push all internal nodes on level 24 and all solid nodes on levels $25 \sim 31$ to level 32. We call this *3-level pushing*. For level 16, our data structure has three arrays: bit map 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: bit map 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 on 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 on level 24 times 2^8.

Lookup Process: Given an IP address a, using $a_{(i,j)}$ to denote the integer value of the bit string of a from the i-th bit to the j-th bit, we first check whether $B_{16}[a_{(0,15)}] = 1$; if yes, then the $a_{(0,15)}$-th node on level 16 is a solid node and thus the next hop for a is $N_{16}[a_{(0,15)}]$; otherwise, then the $a_{(0,15)}$-th node on level 16 is an empty node and thus we need to continue the search on level 24, where the index is computed as $(C_{16}[a_{(0,15)}] - 1) * 2^8 + a_{(16,23)}$. On level 24, denoting $(C_{16}[a_{(0,15)}] - 1) * 2^8 + a_{(16,23)}$ 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 on level 24 is a solid node and thus the next hop for a is $N_{24}[i]$; otherwise, the i-th node on level 24 is an empty node and thus the next hop must be on level 32, to be more precise, the next hop is $(C_{24}[i]-1)*2^8+a_{(24,31)}$. 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 bit maps, next hop arrays, and the chunk ID arrays as separate arrays for generality and simplicity.

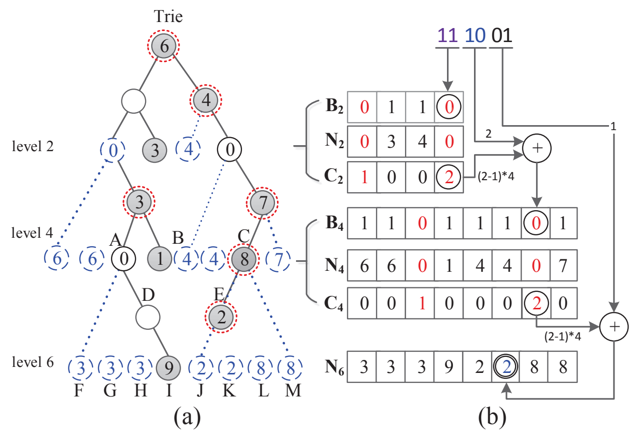

Figure 3: Example data structure of SAIL_L.

Algorithm 2: SAIL_L

```
Input: Bit map arrays: $B_{16}, B_{24}$
Input: Next hop arrays: $N_{16}, N_{24}, N_{32}$
Input: Chunk ID arrays: $C_{16}, C_{24}$
Input: $a$: an IP address
Output: the next hop of the longest matched prefix.
1 if $B_{16}[a >> 16]=1$ then
2     return $N_{16}[a >> 16]$
3 end
4 else if
  $B_{24}[(C_{16}[a >> 16] - 1) << 8 + (a << 16 >> 24)]$ then
5     return
      $N_{24}[(C_{16}[a >> 16] - 1) << 8 + (a << 16 >> 24)]$
6 end
7 else
8     return $N_{32}[(C_{24}[a >> 8] - 1) << 8 + (a\&255)]$
9 end
```

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 $0 \le i \le 2^{16} - 1$, $B_{16}[i] = 0$ 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 $0 \le i \le 2^{16} - 1$, $BCN[i]_{(0,0)} = 1$ 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 $0 \le i \le 2^{16} - 1$, $C_{16}[i]$ 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 $BN_{24}[i] = N_{24}[i]$.

	Finding prefix length	Finding next hop
Prefix length 0~24	B_{16} , C_{16}, B_{24}	N_{16}, N_{24}
Prefix length 25~32	C_{24}	N_{32}

Figure 4: Memory management for SAIL_L.

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.

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 [41]. Second, the modification on the lookup arrays (namely the bit map 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.

4.3 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. Our goal in dealing with multiple FIBs is to build one data structure so that we can perform IP lookup on it for each FIB. Note that our method below can be used to extend SAIL_B and SAIL_U to handle multiple FIBs as well, although for these two algorithms, the FIB update cost is no longer constant for the number of on-chip memory accesses.

Data Structures: Given m FIBs $F_0, F_1, \cdots, F_{m-1}$, first, for each F_i $(0 \le i \le m-1)$, 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 [39]. 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 $0 \leq i \leq m-1$. 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.

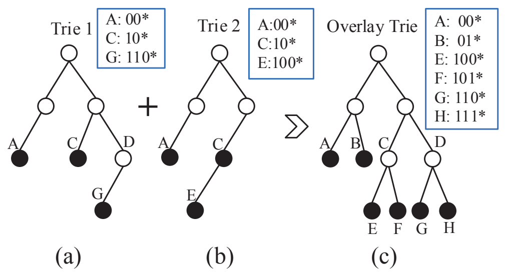

Figure 5: Example of SAIL for multiple FIBs.

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

Two-dimensional Splitting: Regarding memory management, SAIL_M exactly follows the two-dimensional splitting 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 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 a. If it is on 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 on 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.

5. SAIL IMPLEMENTATION

In this section, we discuss the implementation of SAIL algorithms on four platforms: FPGA, CPU, GPU, and many-core.

5.1 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.

5.2 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.

To obtain the number of CPU cycles of SAIL_L and SAIL_M, we reverse engineering our C++ code. The assembly code is shown in Table 2. This table shows that for SAIL_L, given an IP address a, if the next hop of a can be found on level 16, then the lookup needs only 3 CPU instructions; if the next hop of a can be found on level 24, then the lookup needs only 10 CPU instructions. Note that CPU could complete more than one instruction in one cycle. In comparison, for the Lulea algorithm, given an IP address a, if the next hop of a can be found on level 16, then the lookup needs at least 17 $\sim$ 22 CPU cycles; if the next hop of a can be found on level 24, then the lookup needs 148 CPU cycles. This explains why our algorithm is much faster than Lulea.

Table 2: The disassembling code of lookup.

if((H=0 − $BCN_{16}[a \gg 16]$)>0);		
000000013F32589C	movzx	ecx,ax
000000013F32589F	movzx	ebx,byte ptr[r10+rcx*2]
000000013F3258A4	neg	bl
000000013F3258A6	jne	sailtest+147h(013F3258D7h)
else if(H=$BN_{24}[(BCN[a \gg 16] \ll 8) + (a \ll 16 \gg 24)]$);		
000000013F3258A8	movsx	ecx,word ptr[r10+rcx*2]
000000013F3258AD	shl	ecx,8
000000013F3258B0	movzx	eax,byte ptr [rdi+rdx*4+1]
000000013F3258B5	add	ecx,eax
000000013F3258B7	movzx	ebx,byte ptr [rcx+r9]
000000013F3258BC	test	bl,bl
000000013F3258BE	jne	sailtest+147h(013F3258D7h)

5.3 GPU Implementation

We implemented SAIL_L in GPU platforms based on NVDIA's CUDA architecture [30]. 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 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 $32 * k$ ($k \geq 1$) threads to a unique stream. Thus, different streams of $32 * k$ threads can be pipelined, *i.e.*, while one stream of $32 * k$ threads are transferring data from the CPU to the GPU, one stream of $32 * k$ threads are executing their kernel function, and another stream of $32 * k$ threads are transferring data from the GPU to the CPU.

5.4 Many-core Implementation

We implemented SAIL_L on the many-core platform Telera 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.

6. EXPERIMENTAL RESULTS

In this section, we present the experimental results of our algorithms on the four platforms of FPGA, CPU, GPU, and many-core.

6.1 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 not practically feasible to dump the FIB of a tier-1 router to hard disk because of the unbearable overhead incurred on the router. On the server, we use the open source `Quagga` to dump the FIB every hour [2]. We captured real traffic in one of the tier-1 router's interfaces at the interval of 10 minutes per hour between October 22nd 08:00 AM 2013 to October 23rd 21:00 PM.

In addition, we downloaded 18 real FIBs from `www.ripe.net`. Six of them were downloaded at every 8:00 AM on January 1st of each year from 2008 to 2013, denoted by $FIB_{2008}, FIB_{2009}, \cdots, FIB_{2013}$. Twelve of them were downloaded from 12 routers on 08:00 AM August 8 2013, respectively, 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 FIBs downloaded from the 12 routers on 08:00 AM August 8 2013; thus, we guarantee that each prefix has the same opportunity to be hit by the synthetic traffic. We call such traffic *prefix-based synthetic traffic.*

We evaluated our algorithms on four metrics: *lookup speed* in terms of pps (# of packets per second), *on-chip memory size* 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 four well-known IP lookup algorithms: PBF [36], LC-trie [38], Tree Bitmap [40], and Lulea [28]. We implemented our algorithms and these four prior algorithms using C++. 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 a is the 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.

6.2 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 four well known algorithms, because the other four 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 [36], 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.

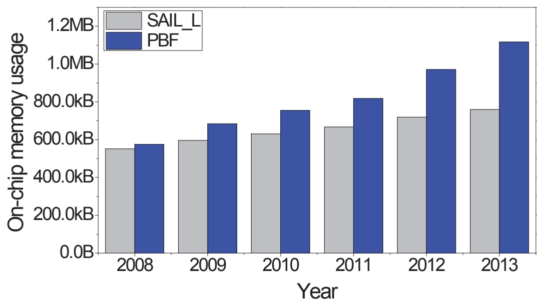

Figure 6: On-chip memory usage over 6 years.

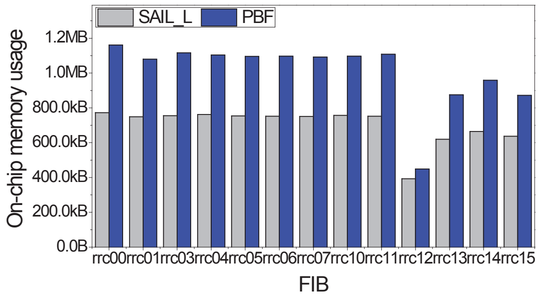

Figure 7: On-chip memory usage of 12 FIBs.

We next evaluate SAIL_L for lookup speed on FPGA platform Xilinx Virtex 7. We did not compare with PBF because [36] 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 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 5.2 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.

6.3 Performance on CPU

We evaluate the performance of our algorithm on CPU platforms in comparison with LC-trie, Tree Bitmap, and Lulea 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, and Lulea algorithms. 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, and 6.62∼7.66 times faster than LC-trie, Tree Bitmap, and Lulea, 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, and 7.61∼7.27 times faster than LC-trie, Tree Bitmap, and Lulea, 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, and 4.24∼4.81 times faster than LC-trie, Tree Bitmap, and Lulea, respectively. Figure 8 shows the lookup speed of these 4 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 4 algorithms with prefix-based synthetic traffic on the 12 FIBs downloaded from `www.ripe.net`. Figure 10 shows the lookup speed of these 4 algorithms with random traffic on FIB_{2013}. From these figures, we further observe that for each of these 4 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.

We now evaluate the FIB update performance of SAIL_L on the data plane. Figure 11 shows the variation of the number of memory accesses per update for 3 FIBs (rrc00, rrc01,

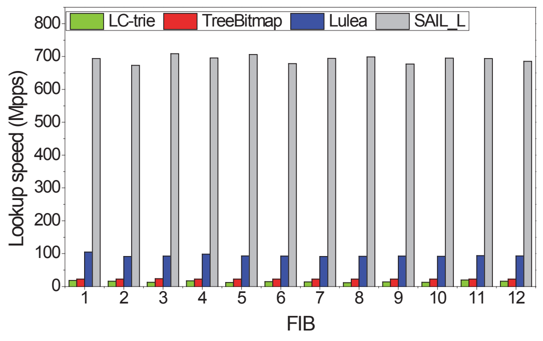

Figure 8: Lookup speed with real traffic and FIBs.

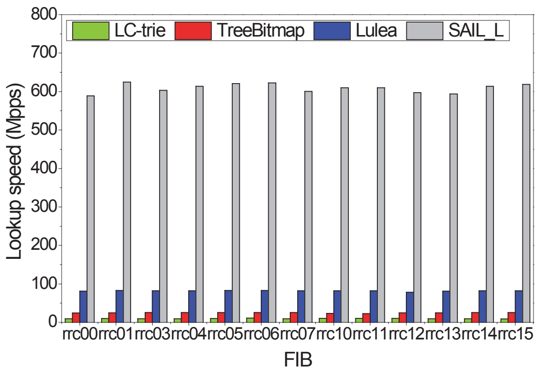

Figure 9: Lookup speed with prefix-based traffic on 12 FIBs.

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 12 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.

6.4 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

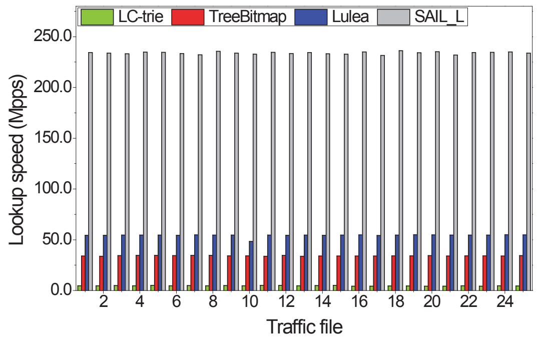

Figure 10: Lookup speed with random traffic on FIB_{2013}.

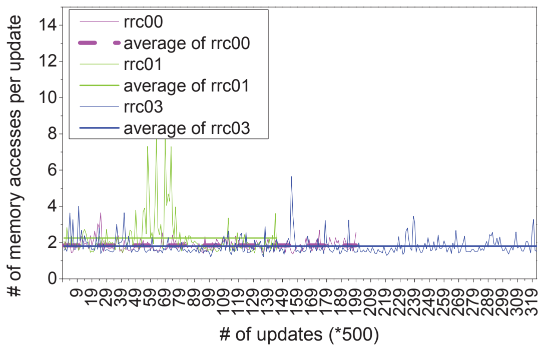

Figure 11: # memory accesses per update.

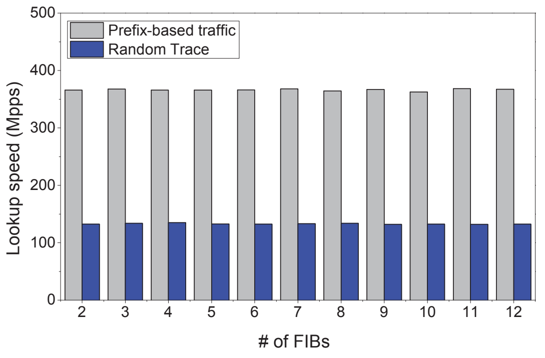

Figure 12: Lookup speed of SAIL_M for 12 FIBs using prefix-based and random traffic.

an NVIDIA GPU (Tesla C2075, 1147 MHz, 5376 MB device memory, 448 CUDA cores). These experiments use the 12 FIBs $rrc00, rrc01$, $rrc03, \cdots rrc07$, $rrc10, \cdots, rrc15$. 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 13 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 14 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 μs. For the traffic size of 60K, SAIL_L has a lookup latency of 110∼152 μs. For the traffic size of 90K, SAIL_L has a lookup latency of 122∼185 μs.

6.5 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 Telera TLR4-03680.

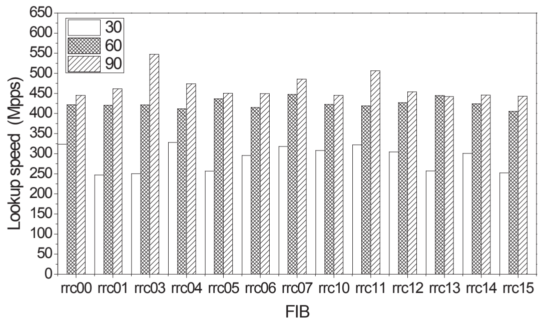

Figure 13: Lookup speed VS. traffic size.

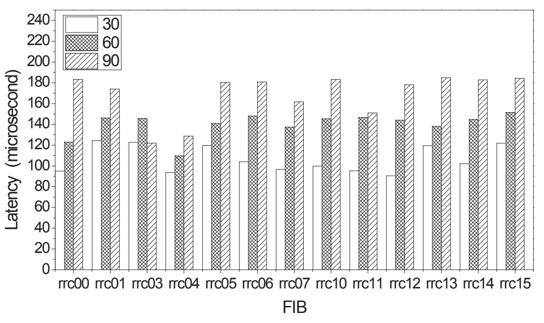

Figure 14: Lookup latency VS. traffic size.

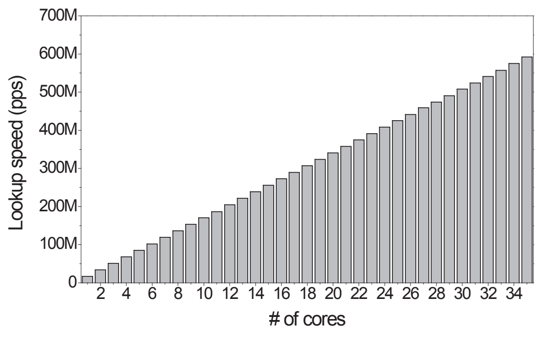

Figure 15: Lookup speed VS. # of cores.

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 15 shows the results on FIB $rrc00$ using prefix-based traffic. We have observed similar results for other FIBs.

7. DISCUSSION

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 on level 48. In other words, we store the bit map and chunk ID arrays of levels 16, 24, 32, 40, and bit map array 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.

8. CONCLUSION

We make three 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 ≤ 24 in on-chip memory of bounded small size. 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 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 three well known IP lookup algorithms (namely LC-trie, Tree Bitmap, and Lulea) that we implemented in [4].

9. ACKNOWLEDGEMENTS

We would like to thank Peng He for implementing the Tree Bitmap algorithm and Chunjing Han, Qinghua Wu and Taihua He for their valuable suggestions. We also would like to thank the anonymous reviewers and our shepherd, Andrew Moore, for their thoughtful suggestions. This work was supported in part by National Basic Research Program of China with Grant 2012CB315801, by National Natural Science Foundation of China (NSFC) with Grants (61133015, 61202489), and by Strategic Priority Research Program of CAS with Grant XDA06010303.

10. REFERENCES

[1] FPGA data sheet [on line]. Available: http://www.xilinx.com.
[2] Quagga routing suite [on line]. Available: http://www.nongnu.org/quagga/.
[3] RIPE network coordination centre [on line]. Available: http://www.ripe.net.
[4] SAIL webpage. http://fi.ict.ac.cn/firg.php?n=PublicationsAmpTalks .OpenSource.
[5] Tilera datasheet [on line]. Available: http://www.tilera.com/sites/default/ files/productbriefs/TILE-Gx8036_PB033-02_web.pdf.
[6] K. Adam and M. Michael. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[7] P. Derek, L. Ziyan, and P. Hang. IP address lookup using bit-shuffled trie. *IEEE Computer Communications*, 2014.

[8] S. Devavrat and G. Pankaj. Fast incremental updates on ternary-cams for routing lookups and packet classification. In *Proc. Hot Interconnects*, 2000.

[9] W. Feng and H. Mounir. Matching the speed gap between sram and dram. In *Proc. IEEE HSPR*, pages 104–109, 2008.

[10] B. Florin, T. Dean, R. Grigore, and S. Sumeet. A tree based router search engine architecture with single port memories. In *Proc. IEEE ISCA*, 2005.

[11] Z. Francis, N. Girija, and B. Anindya. Coolcams: Power-efficient tcams for forwarding engines. In *Proc. IEEE INFOCOM*, pages 42–52, 2003.

[12] R. Gábor, T. János, A. Korósi, A. Majdán, and Z. Heszberger. Compressing IP forwarding tables: Towards entropy bounds and beyond. In *Proc. ACM SIGCOMM*, 2013.

[13] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proc. IEEE INFOCOM*, pages 1240–1247, 1998.

[14] F. Hamid, Z. M. Saheb, and S. Masoud. A novel reconfigurable hardware architecture for IP address lookup. In *Proc. ACM/IEEE ANCS*, pages 81–90, 2005.

[15] S. Haoyu, K. Murali, H. Fang, and L. TV. Scalable IP lookups using shape graphs. In *Proc. ACM/IEEE ICNP*, pages 73–82, 2009.

[16] L. Hoang, J. Weirong, and P. V. K. A sram-based architecture for trie-based IP lookup using fpga. In *Proc. IEEE FCCM*, pages 33–42, 2008.

[17] L. Hyesook, Y. Changhoon, and S. Earl. Priority tries for IP address lookup. *IEEE Transactions on Computers*, 59(6):784–794, 2010.

[18] F. Jing and R. Jennifer. Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. In *Proc. ACM CoNEXT*. ACM, 2008.

[19] Z. Kai, H. Chengchen, L. Hongbin, and L. Bin. A tcam-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Transactions on Networking*, 14(4):863–875, 2006.

[20] S. Keith. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, pages 93–99, 1991.

[21] L. Layong, X. Gaogang, S. Kavé, U. Steve, M. Laurent, and X. Yingke. A trie merging approach with incremental updates for virtual routers. In *Proc. IEEE INFOCOM*, pages 1222–1230, 2013.

[22] H. Lim, K. Lim, N. Lee, and K.-H. Park. On adding bloom filters to longest prefix matching algorithms. *IEEE Transactions on Computers (TC)*, 63(2):411–423, 2014.

[23] M. Mahmoud and M. Massato. A new hardware algorithm for fast IP routing targeting programmable routers. In *Network control and engineering for Qos, security and mobility II*, pages 164–179. Springer, 2003.

[24] W. Marcel, V. George, T. Jon, and P. Bernhard. Scalable high speed IP routing lookups. In *Proc. ACM SIGCOMM*, 1997.

[25] Z. Marko, R. Luigi, and M. Miljenko. Dxr: towards a billion routing lookups per second in software. *ACM SIGCOMM Computer Communication Review*, 42(5):29–36, 2012.

[26] B. Masanori and C. H. Jonathan. Flashtrie: hash-based prefix-compressed trie for IP route lookup beyond 100gbps. In *Proc. IEEE INFOCOM*, 2010.

[27] R. Miguel, B. Ernst, and D. Walid. Survey and taxonomy of IP address lookup algorithms. *Network, IEEE*, 15(2), 2001.

[28] D. Mikael, B. Andrej, C. Svante, and P. Stephen. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM*, pages 3–14, 1997.

[29] A. Mohammad, N. Mehrdad, P. Rina, and S. Samar. A tcam-based parallel architecture for high-speed packet forwarding. *IEEE Transactions on Computers*, 56(1):58–72, 2007.

[30] NVIDIA Corporation. *NVIDIA CUDA C Best Practices Guide, Version 5.0*, Oct. 2012.

[31] C. Pierluigi, D. Leandro, and G. Roberto. IP address lookupmade fast and simple. In *Algorithms-ESA'99*, pages 65–76. Springer, 1999.

[32] W. Priyank, S. Subhash, and V. George. Multiway range trees: scalable IP lookup with fast updates. *Computer Networks*, 44(3):289–303, 2004.

[33] S. Rama, F. Natsuhiko, A. Srinivas, and S. Arun. Scalable, memory efficient, high-speed IP lookup algorithms. *IEEE/ACM Transactions on Networking*, 13(4):802–812, 2005.

[34] P. Rina and S. Samar. Reducing tcam power consumption and increasing throughput. In *Proc. High Performance Interconnects*, pages 107–112, 2002.

[35] H. Sangjin, J. Keon, P. KyoungSoo, and M. Sue. Packetshader: a gpu-accelerated software router. In *Proc. ACM SIGCOMM*, pages 195–206, 2010.

[36] D. Sarang, K. Praveen, and T. D. E. Longest prefix matching using bloom filters. In *Proc. ACM SIGCOMM*, pages 201–212, 2003.

[37] H. Song, M. Kodialam, F. Hao, and T. Lakshman. Building scalable virtual routers with trie braiding. In *Proc. IEEE INFOCOM*, 2010.

[38] N. Stefan and K. Gunnar. IP-address lookup using lc-tries. *Selected Areas in Communications, IEEE Journal on*, 17(6):1083–1092, 1999.

[39] S. Venkatachary and V. George. Fast address lookups using controlled prefix expansion. *ACM TOCS*, 17(1):1–40, 1999.

[40] E. Will, V. George, and D. Zubin. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[41] T. Yang, Z. Mi, R. Duan, X. Guo, J. Lu, S. Zhang, X. Sun, and B. Liu. An ultra-fast universal incremental update algorithm for trie-based routing lookup. In *Proc. ACM/IEEE ICNP*, 2012.

[42] J. Zhao, X. Zhang, X. Wang, and X. Xue. Achieving O(1) IP lookup on gpu-based software routers. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 429–430. ACM, 2010.