Fit the Elephant in a Box - Towards IP Lookup at On-chip Memory Access Speed

Technical Report

ABSTRACT

Fitting large and ever increasing routing tables in small on-chip memory is just like fitting an elephant in a box, which has been considered as impossible. In this paper, we propose the data structure of two *D*imensional *D*ivision *B*loom *F*ilter (D^2BF) that can compactly encode almost all the needed information for performing IP lookup from a FIB in small on-chip memory. With pipelining, we further achieve the throughput of one packet per on-chip memory access. We also propose the data structure of Sentry Bloom Filters (SeBF) to significantly reduce the false positives that stem from the use of Bloom Filters in D^2BF. The probability of accessing off-chip memory to find its next hop is negligible ($1.9 * 10^{-12}$). We simulated our algorithms on FPGA platforms. Experimental results show that we indeed can fit large FIBs in the on-chip memory of today's commodity FPGA platforms, achieving a throughput of 130Gbps.

1. INTRODUCTION

The key challenge of IP lookup is to fit the "elephant" in a "box" - the elephant is the large, yet ever increasing, Forwarding Information Base (FIB, *i.e.*, the IP lookup table), and the box is the small on-chip memory. FIB sizes have been rapidly growing by 15% every year [1], reaching 500,000 prefixes in 2014 [2]. On-chip memory, such as L1, L2, L3 caches on CPU platforms or Block RAM on FPGA platforms, is inherently small in size (in the scale of tens of Mb) and expensive in price. Although small in size, on-chip memory is best for IP lookup because it is tens of times faster than off-chip memory [3].

Although IP lookup has long been a core networking problem and many types of schemes have been proposed (such as TCAM-based, GPU-based, trie-based, and hash-based schemes), the goal of fitting a large FIB in on-chip memory and therefore accomplishing IP lookup in on-chip memory has remained quite distant. Bloom Filters (in short BFs), as space-efficient data structures, have the potential to compactly encode the FIB information needed for IP lookups in the on-chip memory. Dharmapurikar *et al.* proposed to use Bloom Filters in the on-chip memory to encode a FIB [4]; however, their Bloom Filters can only be used to find the length of the longest matching prefix in the FIB for a given IP address. After querying the Bloom Filters in the on-chip memory, their scheme requires to access off-chip memory to find the next hop of the given IP address.

Inspired by this line of thought, in this paper, we propose two-Dimensional Division Bloom Filters (DDBF, in short D^2BF). Contrary to Dharmapurikar *et al.*'s scheme, D^2BF can be used to directly find the next-hop from a simultaneous query of many Bloom Filters. The key insight behind D^2BF is that after the Bloom filter query, a single Bloom Filter reports true, which points to a single next hop. Therefore, with D^2BF, we do not need to explicitly know the longest matching prefix and therefore do not need to store it on-chip, saving the precious on-chip memory. Furthermore, using the techniques from [5], D^2BF can achieve one on-chip memory access per lookup, and almost no off-chip memory access.

However, D^2BF comes with one main challenge. More than one Bloom Filter with different next hops could be matched due to false positives. We address this challenge by introducing two mechanisms, Sentry Bloom Filters and a white list. Throughout this paper, the reader will realise that most of the complexity in the design of D^2BF stems from properly handling the false positives, a consequence of Bloom Filters. While very convenient and space-efficient, Bloom Filters bring complexity into the design of IP lookup, if we want to get very close to IP lookups at on-chip memory access speed.

The contributions of this paper are the following:

- We propose the data structure of D^2BF that can compactly encode almost all the needed information for performing IP lookup from a FIB in small on-chip memory. With pipelining, we further achieve the throughput of one packet per on-chip memory access.
- We propose Sentry Bloom Filters (SeBF) to reduce the overall false positive rate. SeBF reduces the size of the white list, as well as the number of

required Bloom Filters. Given an IP address, the probability of accessing off-chip memory to find its next hop is negligible ($1.9 * 10^{-12}$).

- We validate our approach through extensive experiments based on publicly available routing data.
- We release the source code of D^{2}BF at [6].

The rest of this paper is organized as follows. Section 2 describes our approach, D^{2}BF. We present several approaches to optimize the performance of D^{2}BF in Section 3. We evaluate our approach in Section 4 based on publicly available routing data. Section 5 surveys the related work. Finally, we conclude in Section 6.

2. D^{2}BF

In this section, we present the basics of the D^{2}BF algorithm. We go through all its building blocks, starting from the two dimensional division method, the lookup process, the design of the Bloom Filters, how false positives are handled, as well as the handling of updates.

2.1 Longest Prefix Matching and Tries

The Forwarding Information Base (FIB) is made of a set of prefixes, each prefix having a next-hop. The FIB is usually represented by a trie. We now explain the terminology specific to tries, such as *level*, *solid node*, *empty node*, *leaf node*, *internal node*, based on the example shown in Figure 1. Node A is the root node, it is also a solid node and internal node, its level is 0, and it corresponds to prefix * with next-hop 1. Node E is a solid and leaf node, its level is 2, and it corresponds to prefix 01* with next-hop 3. Node B is an internal node, also an empty node without next-hop (denoted by 0).

For an IPv4 FIB (this paper mainly focuses on IPv4), the length of prefixes ranges from 0 to 32. Given an incoming packet, its destination IP address a will be looked-up in the FIB. There may be more than one prefixes matching a, which is called *prefix overlap*. The Longest Prefix Matching (LPM) will be found, and the packet will be forwarded to the next-hop of the longest matched prefix. For example, in Figure 1, given an IP address 111111, the lookup will match nodes A, C, and K. The next-hop of K will be chosen because the length of the corresponding prefix is the longest.

2.2 Two Dimensional Division

The FIB is a set of prefixes, which we split into many small prefix sets. These sets must satisfy three requirements: 1) only one set reports a match given an IP address; 2) the prefixes of each set have the same next-hops; 3) the prefixes of each set have the same prefix length. Given an IP address to be looked-up, there will be only one set reporting true, so that the next-hop is known upon the set match.

Note that some IP addresses may match no prefix in the FIB when the root node has no next hop. For

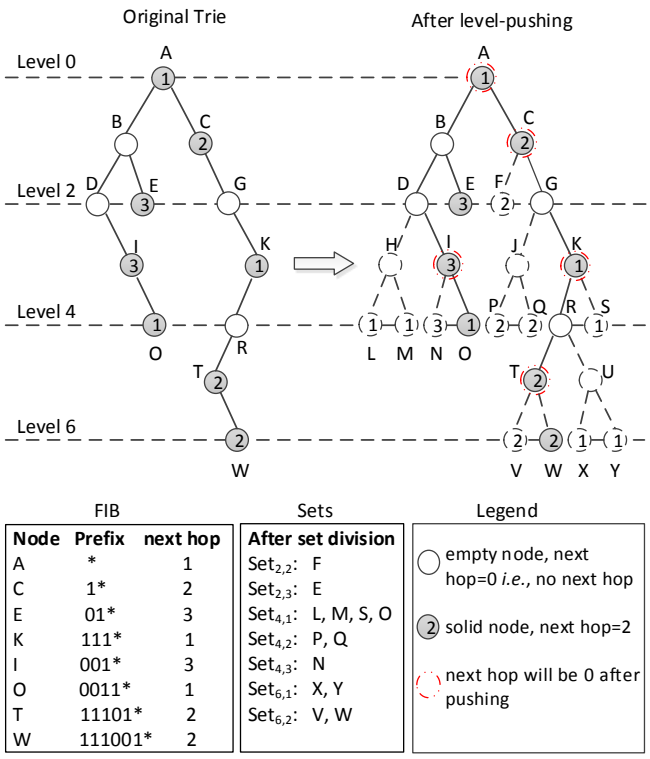

Figure 1: Example of level pushing and set division.

convenience, we assume that the root node always has a next-hop. If this is not the case in the original FIB, we will assign a specific value (such as -1 or 255) to the root node.

To satisfy the first requirement, prefix overlap needs to be eliminated. The reason behind prefix overlap is that there are internal solid nodes in the trie. Therefore, if we can produce a trie with only solid leaf nodes, overlap will be eliminated. One straightforward solution is to rely on leaf pushing [7]. Leaf pushing pushes the internal solid nodes to leaf nodes. For example in Figure 1, node I can be pushed to node N, node T can be pushed to node V. After leaf pushing, only leaf nodes have a next-hop. Therefore, only one prefix (corresponding to a leaf node) will be matched, satisfying the first requirement.

After leaf pushing, we can group the prefixes in two dimensions to satisfy the second and third requirements. The first dimension is the level (*i.e.* prefix length): the solid nodes at the same level of the trie will be put into one set. There will be 32 sets for IPv4 FIBs: S_1, S_2, ..., S_{32}. The second dimension is the next-hop. For each set S_i ($0 \leq i \leq 32$), the prefixes with the same next-hop will be put into one set: $S_{i,j}$ ($0 \leq i \leq 32, 0 < j \leq H$), where H is the number of next hops in the FIB.

After leaf pushing and the above two dimensional division, there will be $32 * H$ sets. Given any IP address, only one set will report a match. The next-hop is found

immediately because the prefixes of any set have the same next-hop.

Level pushing. Thanks to the limited number of next-hops in routers, H is usually smaller than 20. However, $32 * 20$ is still too large. To reduce the number of sets, we propose to use level pushing. As shown in Figure 1, every solid node can be replaced by its two children with same next-hop. The children can also be pushed to the next level. We call the solid nodes *pushed*, hence the name *level pushing*.

Leaf pushing and level pushing can be finished in one traversal of the trie. We give an example to illustrate leaf pushing and level pushing in Figure 1. There are two tries: the left one is the original trie and the right one is the trie after leaf pushing and level pushing. Suppose that we want to push all prefixes into 3 levels: 2, 4, and 6. Node A is pushed to Nodes L, M, and N; Node C is pushed to Nodes F, P, and Q; Node I is pushed to Node N; Node K is pushed to S, X, and Y; Node T is pushed to node V. Nodes E, O and W don't need to be pushed, because they are leaf nodes and already at the pushed level.

Using level pushing, the solid nodes in the trie can be pushed into several specific levels. Note that our set only contains the prefixes corresponding to solid nodes. If the trie is pushed to 4 levels, then the 32*H sets become 4*H sets.

2.3 Set Lookup Algorithm

To summarize the whole method, given a FIB $\mathbb{F}$, we build a trie $\mathbb{T}$, then we carry out leaf pushing algorithm to eliminate overlap, followed by level pushing to generate a trie with limited number of levels: l_1, l_2, ..., l_L. In this way, we divide the FIB into $L * H$ sets $S_{i,j} (i = l_1, l_2, ..., l_L, 0 < j \leq H)$, where i represents the level (prefix length), j represents the next hop, L represents the number of levels after pushing, and H is the number of next hops.

Given an IP address a, we check whether $|a| >> (32 - l_1)$ is an element of $S_{l_1,j} (0 < j \leq H)$, where $|a|$ represents the integer value of a and $>>$ means right SHIFT. At the same time, we check whether $|a| >> (32 - l_2) \in S_{l_2,j} (0 < j \leq H)$, ..., $|a| >> (32 - l_L) \in S_{l_L,j} (0 < j \leq H)$. After these L*H checks, only one set reports true, and the next-hop is the hop ID of the matched set. We denote by $h(S)$ as the hop ID of set S.

2.4 Choosing Optimal Levels

One crucial design choice is the number of levels L. Values of L of 3 or 4 typically lead to a minimal number of prefixes. When L is 3 and 4, there are $C_{32}^3 = 4960$ and $C_{32}^4 = 35960$ possible combinations of levels, respectively. While at first it may appear too time-consuming to compute the optimal levels to be used. As illustrated in Figure 2, most prefixes are at level 24, making it an obvious level choice. Also, 32 should be chosen as another level too. As there are few prefixes of lengths between 25 and 31, these levels should not be chosen. Finally, no prefixes exist with length between 1 and 7, so these levels should not be considered neither. Therefore, as levels 24 and 32 should be chosen, we are left to decide which levels between 8 and 23 should be chosen. In other words, we are left with $C_{16}^1 = 16$ combinations when $L = 3$, and $C_{16}^2 = 120$ combinations when $L = 4$.

Experimental results on 13 FIBs show that: 1) when $L = 4$, the four levels of 18, 22, 24, and 32 achieve the minimum overall number of prefixes in most cases, and 2) when $L = 3$, the three levels of 20, 24, 32 are the optimal levels.

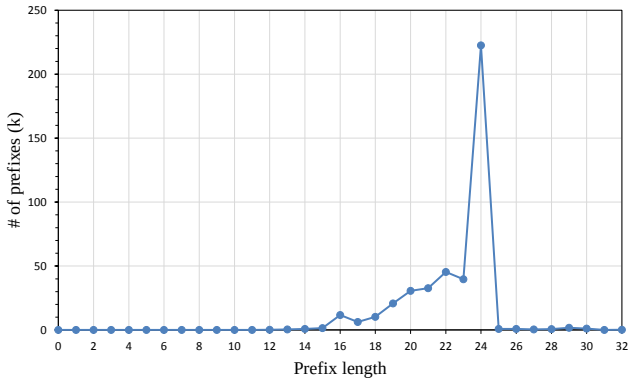

Figure 2: Prefix length distribution.

2.5 Using Bloom Filters

The most important operation in the set lookup algorithm is to judge whether an element belongs to a given set. To achieve this, the naive solution is to traverse the set, with time complexity $O(n)$, where n is the number of elements in the set. Another solution is to construct a hash table, making the average time complexity $O(1)$. However, with the hash table: 1) the worst-case performance cannot be bounded because of hash collisions and; 2) the memory requirements are too large to fit in the on-chip memory if the hash collision probability is to be kept low. Note that perfect hashes cannot be used because of the frequent updates to be made to the FIB. Therefore, in this paper, we propose the use of Bloom filters for the set lookup.

2.5.1 *Bloom filter basics*

A Bloom Filter (BF) is a space-efficient data structure for quickly judging whether an element is a member of a set. It is a bit array of m bits. If an element that actually does not belong to the set is matched is called a false positive (FP). The FP probability of Bloom filters has been extensively studied [8]. Although Bose *et al.* [9] and Christensen *et al.* [10] pointed out that the

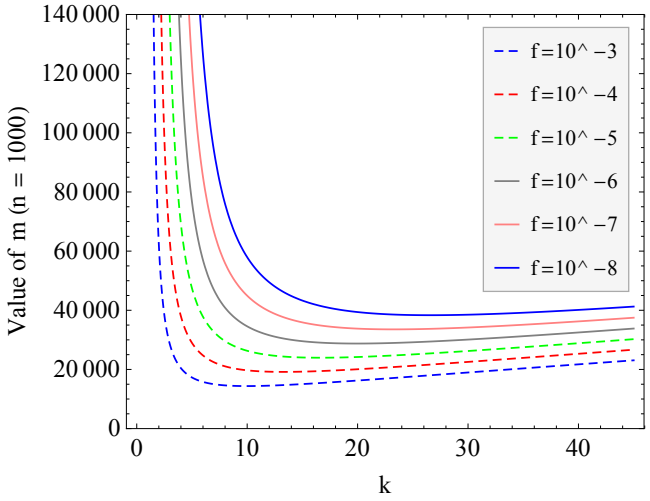

Figure 3: Relationship between m and k when $n = 1000$.

formula regarding the FP probability deduced by Bloom was wrong, the error is so small that it can be ignored when the size of the BF is large. Therefore, we still rely on the original BF formula in this paper.

Suppose that m denotes the size of the Bloom Filter (the number of bits), k denotes the number of hash functions, and n denotes the size of the set, and f represents FP probability.

The FP probability is minimum when

$$k = \ln 2 \frac{m}{n} \tag{1}$$

and the minimum FP probability f is:

$$f = \left(\frac{1}{2}\right)^k \tag{2}$$

According to equations 1 and 2, given a FP probability, we can compute the optimal values of m and k. Figure 3 shows the relationship between m and k when f is given and n is 1000. For example, if f is required to be at its optimal value of 10^{-4}, k should be 13, $m = 19.17n$.

2.5.2 *Two-Dimensional Division Bloom Filters*

As illustrated in Figure 4, our algorithm works as follows:

1. We first build a trie given a FIB. Then we carry out leaf-pushing and level-pushing to produce a trie which has only four levels: 18, 22, 24, and 32. We call this trie *4-level trie* in this paper.

2. We build 4*H BFs, named $BF_{i,j} (i = 18, 22, 24, 32; j = 1, 2, 3..., H)$, with all $0s$ in all BFs.

3. We traverse this 4-level trie, for each prefix $p/l(p) : h(p)$, where $l(p)$ is p's length (level), and $h(p)$ is p's next hop. We insert prefix p into $BF_{l(p),h(p)}$. The lookup table construction is complete.

4. The lookup of D^{2}BF is almost the same as the set lookup algorithm. The only difference is that given an IP address a, we perform the BF query to check whether $a >> (32 - i) \in BF_{i,j}$.

Assuming that the number of next-hops H does not significantly increase in the future, the lookup speed of the D^{2}BF algorithm, as determined by the speed of one query of BF, can be held constant, irrespective of the growth in the routing tables.

2.6 False Positives

There are 4*H Bloom Filters in D^{2}BF, where H is the number of next hops. When selecting the same optimal value of k, all BFs have the same FP probability $f_0 = 0.5^k$. First, we compute the probability that two or more Bloom Filters match f_1:

$$f_1 = 1 - (1 - f_0)^{4H-1} \tag{3}$$

When two or more Bloom Filters with the same next-hop report true, D^{2}BF doesn't need a second confirmation, whose probability f_2 is:

$$f_2 = (C_4^2 f_0^2 (1 - f_0)^{4H-2} + C_4^3 f_0^3 (1 - f_0)^{4H-3} + C_4^4 f_0^4 (1 - f_0)^{4H-4}) * H \tag{4}$$

False positives in D^{2}BF are when two or more Bloom Filters with different next-hops match. In this case, a second confirmation is needed, whose probability is $f_1 - f_2$:

$$f_1 - f_2 = 1 - (1 - f_0)^{4H-1} - H * \sum_{i=2}^{4} C_4^i f_0^i (1 - f_0)^{4H-i} \tag{5}$$

2.7 Update Handling

For the incremental update process, we build one Counting Bloom Filters (CBF) per on-chip BF. CBF uses a counter to replace a bit in the BF, and we store it in off-chip memory. The update mechanism of D^{2}BF is as follows. When inserting a prefix, D^{2}BF adds it to the off-chip CBF and the corresponding on-chip BF, which is as fast as a BF query. When deleting a prefix, the counter of the off-chip CBF decreases by one. When the counter is equal to 0, D^{2}BF sets the corresponding bit of the associated Bloom Filter to 0.

Unfortunately, CBF has false negative when a counter overflows. The probability [11] of false negative is:

$$f_{negative} \leq m(\frac{eln2}{2^c})^{2^c} \tag{6}$$

Where c is the number of bits for a counter, and m is the size of CBF. Typically, four bits is enough for

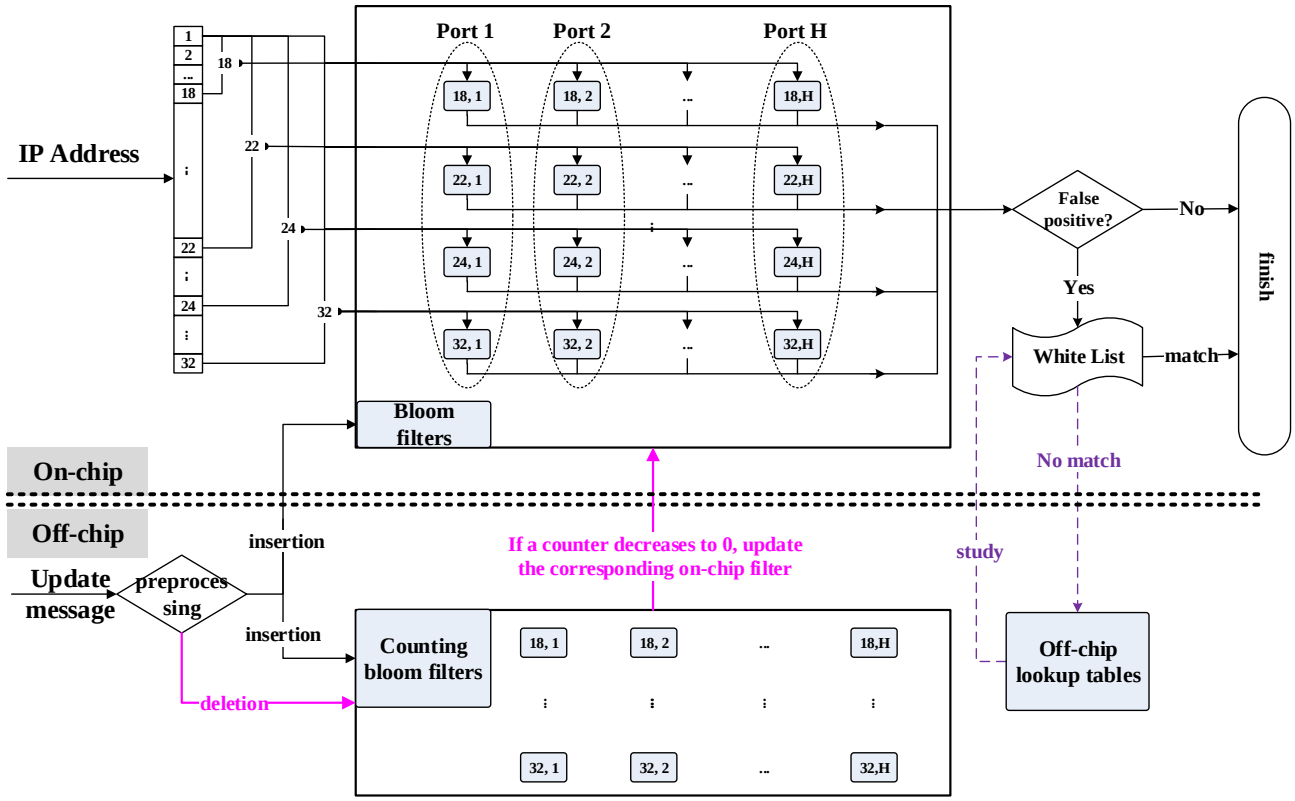

Figure 4: D^{2}BF architecture.

a counter ($f_{negative} \leq 1.37 \times 10^{-15} * m$) . To reduce the false negatives of counting Bloom Filters due to overflows, we can use 8 bits per counter ($f_{negative} \leq 8.3 \times 10^{-547} * m$). Although this means more memory requirements, off-chip memory is plenty (compared to on-chip).

As false negatives may still occur and are a problem for real routers due to the corresponding wrong lookup, we propose a method to prevent false negatives. For each insertion, the counter of the CBF will be incremented by 1. When one counter overflows, we allocate memory with the size of one Bloom Filter to increase the bit size of the overflowing CBF. In other words, as counters overflow, additional memory is allocated to prevent miscounting due to overflows.

3. D^{2}BF OPTIMIZATIONS

Although D^{2}BF algorithm can achieve fast lookup speeds, there is still room for optimizations. In this section, we propose several solutions to optimize the false positives, the number of BFs, the number of memory accesses, and the memory usage.

3.1 White List

When the on-chip memory is large enough, the FP probability of Bloom Filters can be kept low, e.g., below 10^{-8}. When false positives occur, we propose to rely on a white list (WL), as follows:

1. We first build four off-chip tables corresponding to the 4-level pushed binary trie. For levels 18, 22, and 24, we build corresponding non-sparse arrays of size $2^{18}, 2^{22}$ and 2^{24}, respectively. Each element of the array is 8 bits long and stores the next-hop. Then we build hash tables for level 32. Because 2^{32} is very large, we rely on a collision link table to handle hash collisions. As the worst-case collision is not bounded, an alternate option is to use two tables for level 32: one offset table at level 24 and one next-hop table at level 32 to eliminate hash collision, at a cost of 2 off-chip memory accesses.

2. If more than one Bloom Filter is matched, it means a false positive has occurred. In this case, the IP address is searched in the WL. If matched, the lookup is finished; otherwise, we go to the next step.

3. The prefix length of the matched Bloom Filters is known. Thus we only check one or two of the four off-chip tables. After the correct next-hop is found, the matched prefix and the corresponding next-hop are inserted into the WL.

Table 1: The structure of WL

Prefixes	Next hop
111*	2
101*	2
1011*	2
010*	4
01110*	4

Using on-chip memory, the overall false positive is very low, such as $10^{-7}, 10^{-8}$. Note that the looked up IPv4 address has less than $2^{32} \approx 4.3 \times 10^9$ values, because some addresses are reserved or private. Based on this observation, we can study the WL in advance by traversing the 4G IP address space. Experimental results show that the size of WL is very small. Indeed, the WL is so small that can be held in the on-chip memory (see Figure 20), thus we can use an independent on-chip memory for the WL. The Bloom Filter stage and the WL stage can be looked-up using pipelining, or in parallel.

The worst case of the WL lookup happens when multiple Bloom Filters with different next hops match, but the WL does not contain the prefix due to a FIB updates. In this case, we will lookup the off-chip memory hash tables, and in the worst case require 2 off-chip memory accesses. Note that the WL needs to be updated when an update message arrives.

There are many ways to lookup the WL, because of its small size. One way is to rely on a small TCAM to store the prefixes of the WL. Another is to use hash tables.

After analysing the characteristics of the WL, we propose a new lookup scheme. A sample of WL is shown in Table 1. To achieve high lookup performance, we group the prefixes of the WL according to their next-hop. For example, if two Bloom Filters with next-hops 2 and 4 are matched, the WL algorithm searches the prefixes with next-hop 4 using binary search, because the prefixes with the next-hop 4 are fewer than the ones having next-hop 2. If matched, the incoming packet should be forwarded to port 4; otherwise to port 2. In this way, the number of memory accesses of the WL can be considerably reduced. Experimental results show that assuming a relatively high probability of FP of 10^{-5}, the average number of memory accesses in the WL is only 4, and the number of entries in the WL is 243. Therefore, the WL algorithm can cope with the false positives of Bloom Filters at the cost of a little additional memory overhead and a few additional memory accesses.

3.2 Using Sentry Bloom Filters

To mitigate the false positives of BFs at multiple levels of the trie, we introduce the notion of *gate sentry*. A gate sentry helps us to ascertain that whether we need to check the BFs at the levels below. If a gate sentry for a given level is certain that *the incoming IP address matches no internal node at the given level*, then there is no need to check the BFs at levels below, and the BF queries terminate. Otherwise, we need to query the BFs at levels below.

3.2.1 Level Gate Sentry

In D^{2}BF, all 4*H BFs need to be checked for each lookup. This incurs a relatively large overall FP probability, and therefore a relatively large WL. To reduce the overall FP probability and the size of the white list, we propose to add a gate sentry that tells whether we should check the BFs at the levels below.

As shown in Figure 5, after pushing, D^{2}BF builds 4*H BFs. We add a gate sentry at levels 18, 20, 24, and 32. The lookup process becomes: given an IP address a, first check the BFs at level i ($i = 18, 20, 24, 32$) before going to the next level. When a given level is considered, a is checked by the gate sentry at that level. If the gate sentry answers "Yes", then go to the next level; otherwise, the BF queries end.

3.2.2 Sentry Bloom Filter

The rationale behind the gate sentry is as follows. Given an IP address a and level i, a could match no node at level i, a could also match an internal or leaf node at level i. If and only if a matches an internal node at level i, the BFs at levels below need to be checked. Therefore, a gate sentry should know whether an IP matches an internal node at the given level. Two techniques are possible to implement a gate sentry:

1. Building a bitmap of size 2^i at level i, which is initially all $0s$. We use 1 to represent internal nodes. This scheme works well only when i is small, because it needs too much memory when i is large.

2. Building Sentry BFs. We build a BF for the internal nodes at each level. This scheme works better than bitmaps when i is large. Therefore, we can combine these two techniques to implement gate sentries in our work.

Thanks to Sentry BF, we can use 3-level pushing tries: level 20, 24, and 32, while holding the entire FIB in on-chip memory. Specifically, we add a bitmap at level 20 and a Sentry BF at level 24.

Level 20 data structure. As shown in Figure 6, we combine the next hops at level 20 and $Bitmap_{20}$ into one array. The size of this array is 2^{20}, each element has 8 bits and is a next-hop or zero. If it is a next hop, it corresponds to a solid leaf node in the 3-level pushed trie; if it is 0, it corresponds to an internal node with no

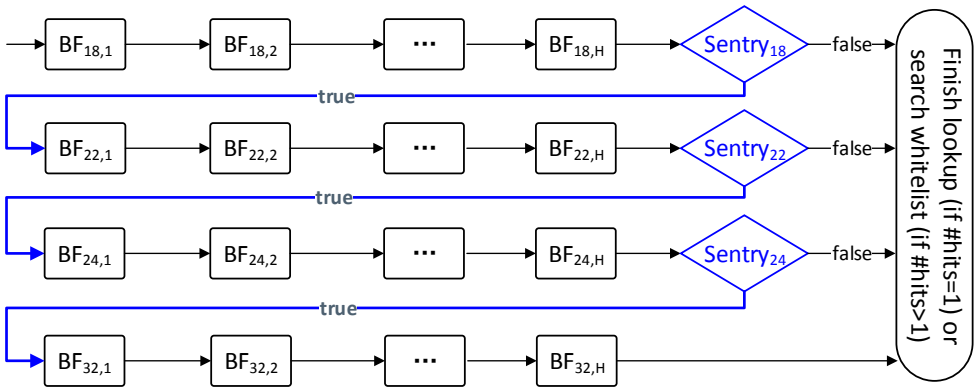

Figure 5: Sentry Architecture.

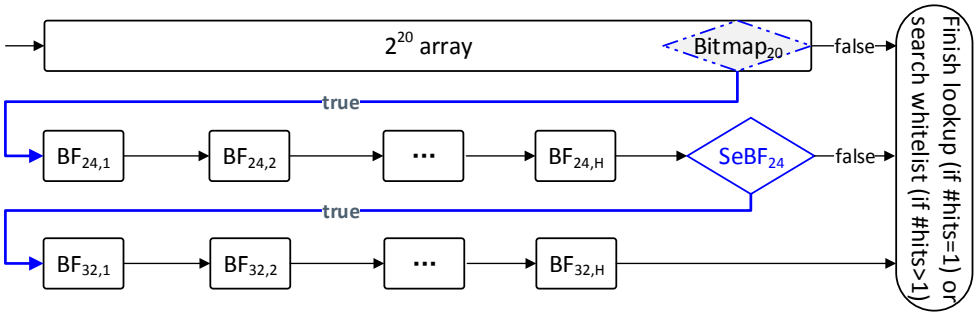

Figure 6: Sentry Bloom filters Architecture.

next hop. Using this method, if an IP address matches a solid leaf node at level 20, the lookup ends. In other words, the gate sentry at level 20 has no false positive, and will never let any unnecessary lookup propagate to level 24.

Level 24 data structure. We build a Sentry BF at level 24, namely $SeBF_{24}$. We insert all the corresponding prefixes of internal nodes at level 24 into $SeBF_{24}$. In this way, if the IP address matches a solid leaf node at level 24, the $SeBF_{24}$ will report false regardless of false positives, and we do not need to check the BFs at level 32. However, if a leaf node at level 24 is indeed matched, the $SeBF_{24}$ could report true due to the false positive of $SeBF_{24}$. In other words, the gate sentry at level 24 could let unnecessary lookups propagate to level 32 with a small probability, which is equal to the FP probability of $SeBF_{24}$.

For level 24, the size of Sentry BF is $nk/ln2$, where k is the number of hash functions used, and n is the number of internal nodes at level 24. Based on our experiments, n is less than 4000. If $k = 10$, the memory usage of SeBF is then 7.04KB; if $k = 19$, the memory usage is 13.4KB.

3.2.3 White List and False Positive

When using SeBF, there are H BFs at level 24, each of them uses k_1 hash functions. At level 32, there are H BFs, each of them uses k_2 hash functions. The single SeBF at level 24 uses k_3 hash functions. Suppose there are n_1 and n_2 solid nodes at levels 24 and 32, respectively, and n_3 internal nodes at level 24. Then we compute the size of the white list W_l after traversing all the IPv4 address space in the following.

DEFINITION 3.1. ***Definition 1.*** *Given a BF using k hash functions, when $m/n = kln2$, the FP probability is optimal as 0.5^k. We define f(1,k)=0.5^k, where "1" means there is only 1 BF.*

DEFINITION 3.2. *Given H BFs. For each BF, it uses k hash functions, and $m/n = kln2$. When querying an element, which belongs to one BF, the false positive is that more than one BF report true. We define the overall FP probability as $f(H-1,k) = 1-(1-0.5^k)^{H-1}$.*

Based on these two definitions, next, we compute the size of WL at level 24 and 32, respectively.

1) First, we compute the size of WL at level 24. There are two cases.

Case I: $SeBF_{24}$ reports false (the probability is $1 - f(1,k_3)$). This means no needs for checking BFs at level 32. Then when more than 1 BF on Level 24 report true, it is false positive. The FP probability of such case is: $(1-f(1,k_3))*f(H-1,k_1)$. The number of combinations is m_1.

Case II: $SeBF_{24}$ reports true (the probability is $f(1, k_3)$). There are also two cases.

Case II.1: The BFs at level 32 reporting true are all false positives, and the true matched BF is at level 24. In this case, the FP probability is $f(1, k_3) * f(H-1, k_2) * (1-f(1, k_1))^{H-1}$. Here is $f(H-1, k_2)$, but not $f(H, k_2)$, because the BF with the same next hop with the true BF at level 24 should not be considered. The number of combinations is $m_1 * 256$.

Case II.2: No BF at level 32 reports true, and more than one BF at level 24 report true. In this case, the FP probability is $f(1, k_3) * (1 - f(1, k_2))^H * f(H-1, k_1)$. The number of combinations is $m_1 * 256$.

2) Second, we compute the size of WL at level 32. There are two cases.

Case I: No BF at level 24 reports true. Then more than 1 BF at level 32 report true. In this case, the FP probability is $(1-f(1, k_1))^H * f(H-1, k_2)$. The number of combinations is m_2.

Case II: One or more BFs at level 24 report true. Because we are computing the size of WL at level 32, at least 1 BF at level 32 will report true. In this case, the FP probability is $f(H, k_1)$. The number of combinations is m_2.

In sum, the overall size of WL is:

$$\begin{aligned} W_l &= (1 - f(1, k_3)) * f(H-1, k_1) m_1 \\ &+ f(1, k_3) * f(H-1, k_2) * (1 - f(1, k_1))^{H-1} * 256 m_1 \\ &+ f(1, k_3) * (1 - f(1, k_2))^H * f(H-1, k_1) * 256 m_1 \\ &+ (1 - f(1, k_1))^H * f(H-1, k_2) * m_2 \\ &+ f(H, k_1) * m_2 \end{aligned} \tag{7}$$

Obviously, given H BFs, when two or more BFs report true, it is false positive, so the overall FP probability is roughly the probability that two BFs report true, because the probability that three or more BFs report true is much smaller than that of two. Therefore, we have $fp(H-1, k) = 1 - (1 - f(1, k))^{H-1} \approx (H-1) * f(1, k)$, and the size of WL is reduced to:

$$\begin{aligned} W_l =& 256 n_1 * 0.5^{k_3} (1 - 0.5^{k_2})^H (1 - (1 - 0.5^{k_1})^{H-1}) \\ &+ n_1 (1 - 0.5^{k_3}) (1 - (1 - 0.5^{k_1})^{H-1}) \\ &+ 256 n_1 * 0.5^{k_3} (1 - 0.5^{k_1})^{H-1} (1 - (1 - 0.5^{k_2})^{H-1}) \\ &+ n_2 (1 - (1 - 0.5^{k_1})^H) \\ &+ n_2 (1 - 0.5^{k_1})^H (1 - (1 - 0.5^{k_2})^{H-1}) \end{aligned} \tag{8}$$

The overall FP probability f_{SeBF} is

$$f_{SeBF} = \frac{W_l}{2^{32}} \tag{9}$$

The overall memory occupation M is

$$M = 2^{20} * 8 + \frac{n_1 k_1}{ln2} + \frac{n_2 k_2}{ln2} + \frac{n_3 k_3}{ln2} \tag{10}$$

It is unimportant for the reader to understand the derivations. However, we will revisit the question of the values of k_1, k_2, and k_3, n_1, n_2, and n_3 and their implications in the evaluation section (Section 4).

When update messages arrive, first update the trie and BFs, we still need to check and update the white list (WL). After BFs are updated, when false positive occurs, the WL might report a miss. Then we need to find the true next hop in off-chip tables. The probability of WL miss is studied in Section 4.6.

3.3 Reducing Number of Memory Accesses

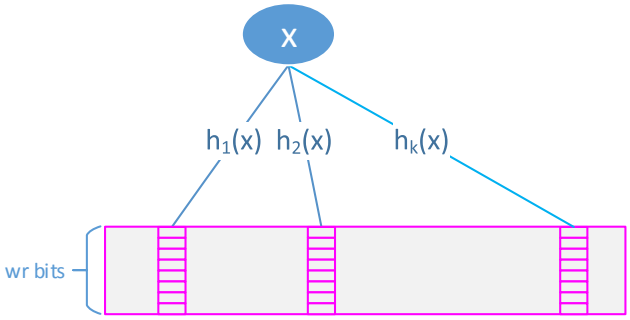

Figure 7: One memory access Bloom Filter for FPGA.

The membership query of Bloom Filters begins with computing k values ranging from 1 to m using k independent hash functions, where m is the size of the Bloom Filter. If the on-chip memory supports k reading ports, one Bloom Filter query can be accomplished in one memory access. PBF [4] assumes that the fabrication of 6 to 8 read ports for an on-chip Random Access Memory (RAM) is possible. Actually, although it is possible to make on-chip memory with 3 or more ports, the widely used option is 2-port memory. Therefore, when k is large, one memory access claimed by PBF cannot be guaranteed.

Fortunately, one memory access can be achieved by the approach proposed in [5], which confines the hash values of k hash functions to r words. If all hash values can be confined to a single word ($r = 1$), one memory access is enough. However, relying on a single word increases the FP probability. The larger the value of r, the smaller the FP probability. For CPU implementations, one memory access can only read 32 or 64 bits. This is not large enough to achieve a small FP probability.

Fortunately, we can read as many bits as we want using the on-chip memory of FPGA. The scheme works as follows. As shown in Figure 7, we split the Bloom Filter into partitions, each with the size of r words. For each BF, the number of partitions is $\frac{m}{wr}$, where w is the

length of a word. If r is too small, the k hash values will be confined to a very small range wr, creating unbalanced distributions of $1s$ in the BF. The sacrifice in false positives will not be negligible. If r is too large, wr will be large, requiring too many address lines in the on-chip memory, decreasing the frequency at which the FGPA operates. Therefore, a good trade-off is necessary: $\frac{m}{wr} >> k$. Typically, we choose r=128 bits and 256 bits to achieve almost no additional false positive.

In addition, for PBF and D^{2}BF, there are multiple BFs which need to be queried. To achieve one clock cycle per lookup, multiple independent memories are needed, which can be easily implemented in on-chip memory [12]. In this way, parallel or pipelined membership queries of BFs can be completed in one clock cycle.

3.4 Using ONRTC

In order to achieve an empty intersection of the Bloom Filters, prefix overlap should be eliminated before our two-dimensional division. One solution is to rely on leaf-pushing [7]. Unfortunately, leaf-pushing causes a significant expansion of the routing table size, and indirectly requires larger Bloom Filters. To save the precious on-chip memory, we adopt the previous proposed ONRTC algorithm [13] instead of leaf-pushing.

The ONRTC algorithm constructs non-overlappping FIBs, and is provably optimal in terms of number of prefixes. The method is conceptually made of two steps:

1. Carry out the leaf-pushing method, pushing all solid nodes as leaf nodes;
2. If two sibling solid nodes have the same next-hop, these two nodes will be replaced by their parent node with the same next hop.

Note that a "Old port" is kept for each node during pushing to facilitate fast incremental updates. Note that these two steps are finished in one traversal of the trie. Experimental results show that about 40% on-chip memory is saved when leaf-pushing is replaced by the ONRTC algorithm [13].

4. EXPERIMENTAL RESULTS

4.1 Data Set

In this section, we study the behavior of our approach based on publicly available FIBs. The FIBs and updates are taken from RIPE RIS [2]. We rely on the full RIBs and updates of 13 route collectors from August 8 2013, the IDs are shown in Table 2. We use the next hop AS from the BGP routes as the next hop in the FIB.

4.2 Leaf-pushing vs. ONRTC

First, we compare the two considered prefix overlap elimination algorithms: leaf-pushing and ONRTC. Even though we carried out the comparison on the 13 collectors, the results are similar across them so only show the results for rrc00. We show in Figure 8 for each prefix length the number of prefixes left after overlap elimination, as well as the original number of prefixes. Note that we do not show prefix lengths smaller than 12 as there are almost no prefix at these lengths. We observe that ONRTC performs much better than leaf-pushing. Therefore, unless stated otherwise, all following experiments are based on the FIBs after having been processed by ONRTC.

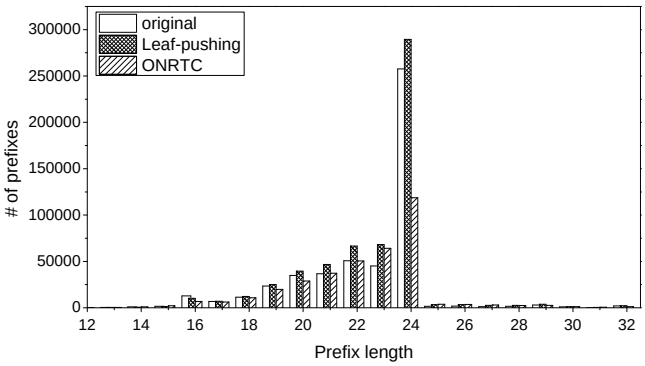

Figure 8: Comparison of leaf-pushing and ONRTC.

4.3 Pushing to 3 vs. 4 Levels

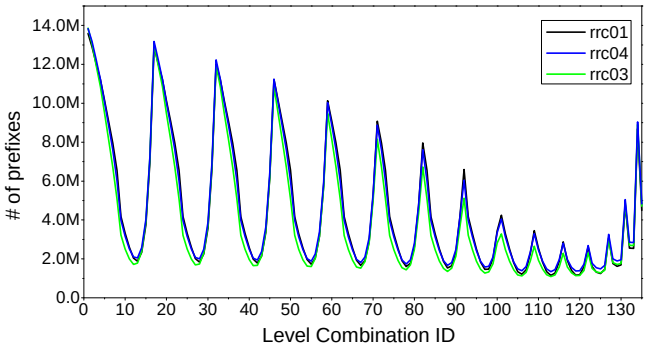

Figure 9: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

As already discussed, levels 24 and 32 have to be chosen, while the other two levels in the range between 8 and 23 are still open. We consider all possible pairs from $< 8, 8 >$, $< 8, 9 >$, ..., to $< 22, 23 >$, $< 23, 23 >$. We assign consecutive id's to each combination and show in Figure 9 $\sim$ 12 the total number of prefixes at the four levels (after overlap elimination) for each pair id. The three curves in Figure 9$\sim$ 12 correspond to FIBs from different collectors. We observe that all FIBs have similar behavior, though the total number of prefixes varies slightly. The peaks on the curves happen when the two levels of the pair are the same, as expected, due to the

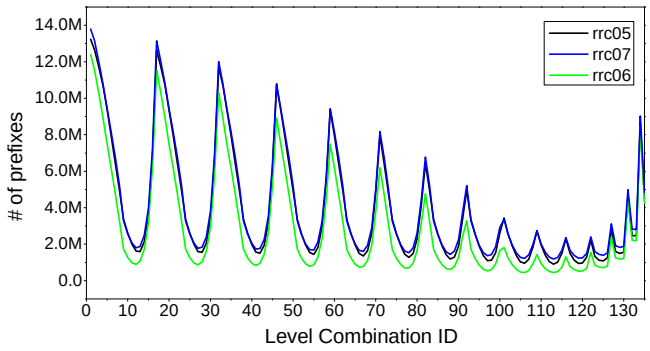

Figure 10: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

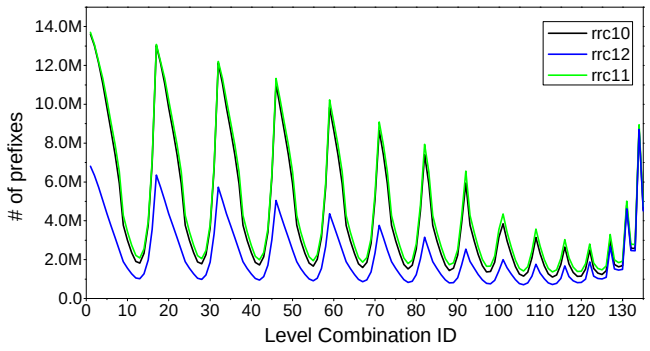

Figure 11: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

redundancy of these two levels. The dips occur when level 22 is considered in the pair, the optimal pair being $< 18, 22 >$. Note that results are similar when considering other route collectors, with the levels 18, 22, 24 and 32 being optimal in most cases and suboptimal in other cases. Note that using the suboptimal set of levels results in little difference for the system performance.

When pushing to 3 levels, the similar results are shown in Figure 13 $\sim$ 16. Similar again, level 24 and 32 are chosen, then we plot the overall memory usage with

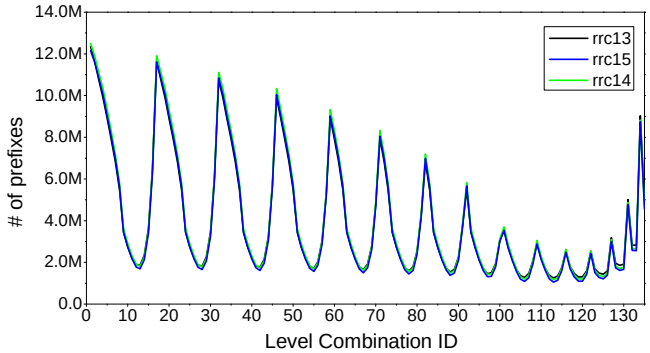

Figure 12: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

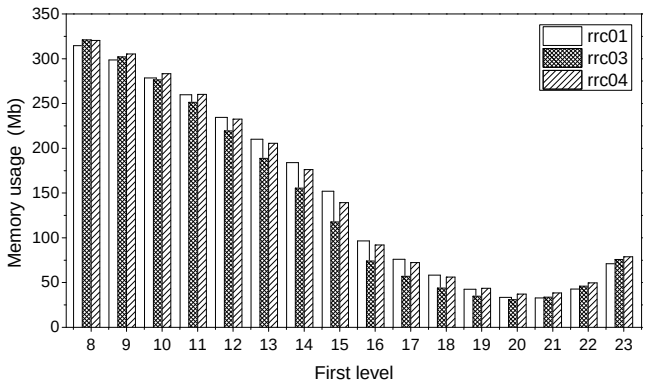

Figure 13: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

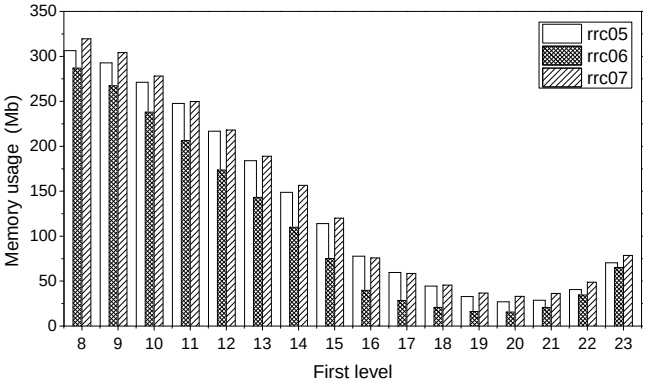

Figure 14: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

different first level ranging from 8 to 23. These results show that different FIBs have similar trends, and 20, 24, and 32 are mostly the optimal combination. Therefore, when pushing to 3 levels, we choose levels 20, 24, and 32.

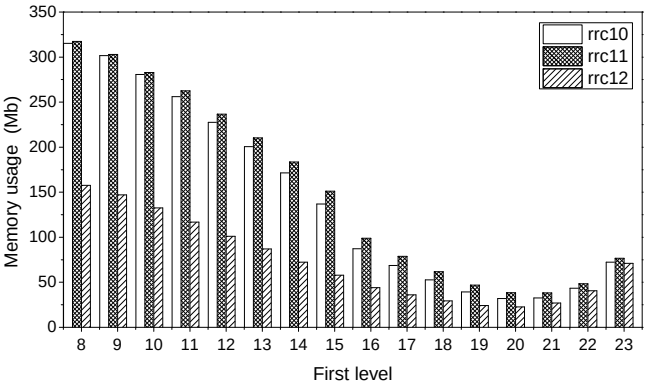

Figure 15: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

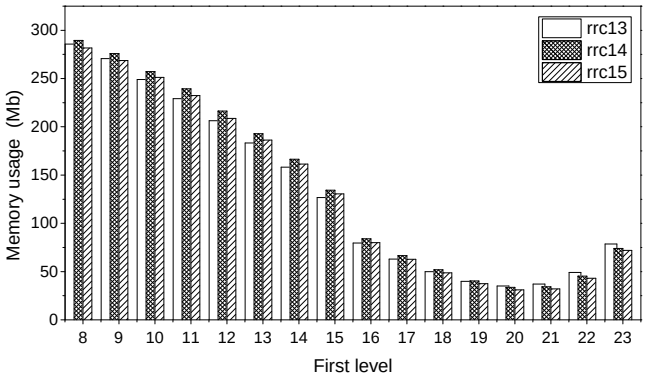

Figure 16: # of prefixes across 4 levels for different level combinations in the 8 to 23 range.

The prefix distribution after pushing is shown in Figure 17, for each of the 4 levels. We observe some diversity in the number of prefixes at different levels. Note that rrc00 contains many more prefixes than the other collectors, and therefore its prefix distribution is quite different from other collectors and not representative of most FIBs. Quite a few collectors surprisingly have a significant fraction of the prefixes at level 32, which we may not expect given the limited number of prefixes whose length is larger than 24. This is explained by level pushing that does create a significant number of prefixes at the last level, independent from the original prefix length distribution in the FIB.

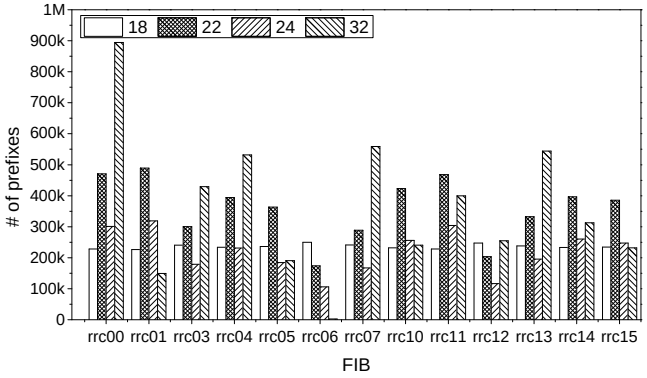

Figure 17: Prefix distribution after pushing to 4 levels.

We now show the number of prefixes after leaf-pushing (using ONRTC) using 3 levels instead of 4. When using 3 levels, the optimal levels are typically 20, 24, and 32. Again, we rely on the same 13 route collector FIBs and show the prefix distribution in Figure 18. We immediately observe the large number of prefixes at the first level (20), close to one million prefixes. The second level (24) has less prefixes than the first, with between 300k and 900k prefixes. Level 3 (24) has between 150k and 500k prefixes. Overall, we see that relying on 3 levels instead of 4 leads to more total prefixes (across all route collector FIBs).

Table 2: # of prefixes and memory usage when pushing to 3 and 4 levels.

	# of prefixes (M)		Memory usage (Mb)	
ID	**3 levels**	**4 levels**	**SeBF**	**D^2BF**
rrc00	2.66	1.81	49.75	50.47
rrc01	1.99	1.13	33.45	25.88
rrc03	1.91	1.10	30.88	28.51
rrc04	2.16	1.33	37.20	35.04
rrc05	1.75	0.93	27.04	21.74
rrc06	1.29	0.51	15.63	9.84
rrc07	2.00	1.20	33.09	32.43
rrc10	1.94	1.10	32.04	26.24
rrc11	2.20	1.34	38.41	33.65
rrc12	1.58	0.78	22.57	19.24
rrc13	2.08	1.25	35.07	33.42
rrc14	2.01	1.15	33.59	28.29
rrc15	1.90	1.05	31.05	25.00

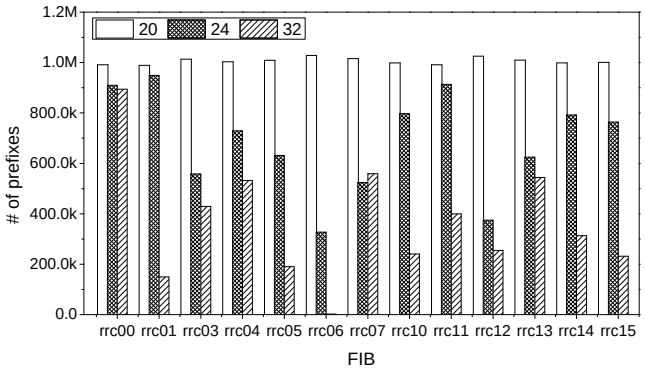

Figure 18: Prefix distribution after pushing to 3 levels.

However, the total number of prefixes is not the only relevant metric for us. Indeed, the total memory consumption depends on the total number of prefixes. However, when using Sentry BFs, the overall false positive is significantly reduced, then we can use fewer hash functions, thus needs much smaller memory usage for each BF. Table 2 compares the total number of prefixes and the memory usage for 3 and 4 levels. We observe that despite the significant difference in total prefixes between 3 and 4 levels, the total memory usage is quite similar, well below the on-chip capacity of today's FPGA (66Mb).

4.4 Cost of Sentry BF

When there are few internal nodes at level 24, our Sentry BF mechanisms (see Section 3.2) is actually very efficient in terms of memory usage. To support

our claim, we plot the number of internal nodes in Figure 19. We observe that the number of internal nodes ranges from 10 to 3493, with an average of 1424. Considering the worst case of 3493, even if we use 20 hash functions for the SeBF, the memory usage is only $20 * 3493/0.7 = 12$KB, which is only about 0.1% of the on-chip memory of a FGPA.

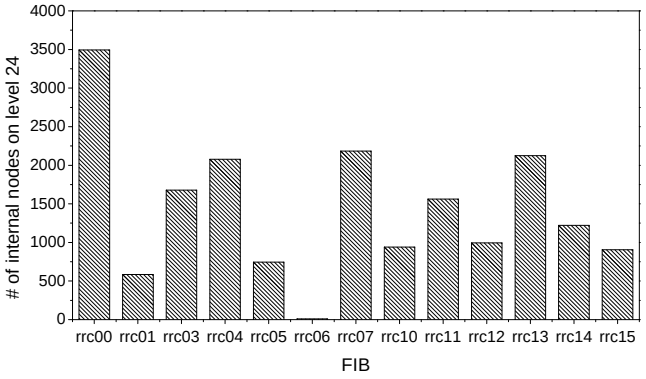

Figure 19: Number of internal nodes at level 24.

4.5 White List Size

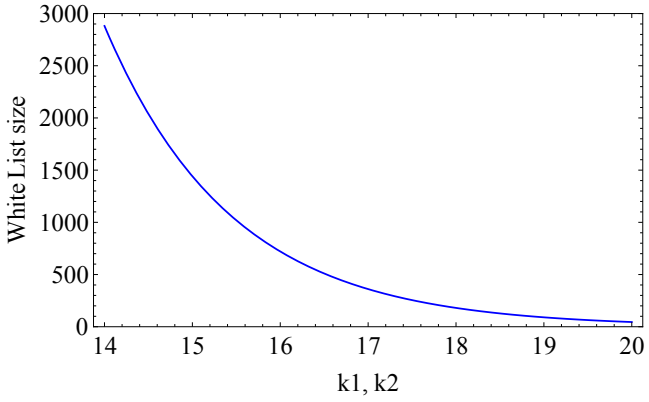

Figure 20: Size of white list as a function of $k_1 = k_2 = k$ using SeBF (with $k_3 = 14$).

We now study the size of the white list, aimed at handling the false positives of BFs. We use the FIB of rrc00 and push it into 3 levels (20, 22, and 24), and plot the expected size of the white list for SeBF in Figures 20 and 21. The expected size depends on the FP probability, which we computed in Section 3.2 as a function of the values of k_1, k_2, and k_3. We study the relationship between the values of k_1, k_2, and k_3, and the expected size of the white list, according to formular 8. We aim to find the values of k_1, k_2, and k_3 that make the size of the white list as small as possible. Let us first concentrate on k_1 and k_2. As shown in Figure 20, if we set $k_1 = k_2 = k$ (with $k_3 = 14$) equal or larger than 17, the size of the white list is smaller than 400. In Figure 20,

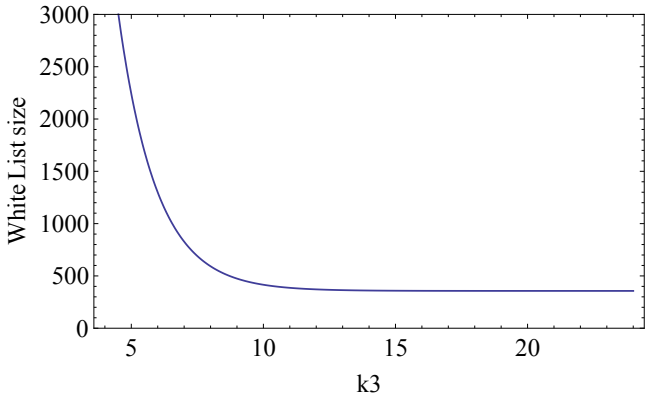

Figure 21: Size of white list as a function of k_3 using SeBF (with $k_1 = k_2 = 17$).

we show the size of the white list as a function of k_3 when $k_1 = k_2 = 17$. Interestingly, we observe that when $k_3 > 10$, the size of white list does not decrease and stays at about 400. From this section, we recommend the following values: $k_1 = k_2$ around 17 and $k_3 > 11$.

4.6 Updates

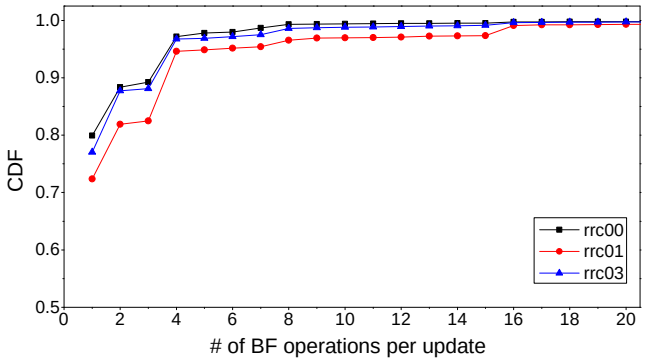

Figure 22: Distribution of number of Bloom filter operations per BGP update.

As most existing FPGA's on-chip memory is 2-port, lookups and updates can be performed simultaneously. Here, we record the number of on-chip BF operations (BF deletions and insertions) per BGP routing update, using a 3 days long trace. Figure 22 shows the distribution of the number of operations across the trace. We see that more than 70% of BGP updates incur a single BF operation, and more than 90% of BGP updates require less than 4 BF operations.

When BGP update messages arrive, we need to update the trie and the BFs, as well as the white list. After the BFs are updated, when a false positive occurs, the white list might actually report a miss. We now attempt to estimate the probability of a white list miss. Suppose there are 500 entries in the white list, and there are 2M

prefixes after pushing to 3 levels. According to updates reports from AS6447 [14], the average number of BGP prefix updates per second is 4.08. Suppose that one update incurs the worst-case of 8 prefix updates in the 3-level pushed trie. The probability of accessing the white list is $500/2^{32}$. The probability of updating a prefix in the white list is therefore $500 * 4.08 * 8/(2 * 10^6)$. The probability of accessing an updated prefix in the white list is $500 * 4.08 * 8/(2 * 10^6 * 2^{32}) = 1.9 * 10^{-12}$.

4.7 FPGA Simulation

We use the Xilinx ISE 13.2 IDE to simulate system throughput, and use the XILINX Virtex-7 FPGA (XC7VX1140T) as the target platform. It has 1880 dual-port block RAMs, each block storing 36 Kb, a total of 66Mb block RAM (on-chip memory). Block RAMs are fundamentally 36 Kb in size, but each block can also be used as two independent 18 Kb blocks. Two adjacent 36 Kb block RAMs can be configured as one cascaded 64kb dual-port RAM without any additional logic.

We take the FIB of rrc04 to simulate the system throughput. After pushing to 3 levels, there are 22 BFs and 1 SeBF. We set $k_1 = k_2 = 17$ and $k_3 = 14$. Each BF is assigned to one stage using one independent block memory. The full array of level 20 is also stored in one independent block memory. Using pipelining, the simulation results shows a minimum clock period of 2.457 ns, *i.e.*, a maximum frequency of 407.013MHz. It means the throughput of our SeBF solution on FPGA platform can reach 407 Mpps.

5. RELATED WORK

IP lookups is a well-researched topic because of its fundamental role in the Internet. In this section, we review the dominant algorithms, which can be classified into six categories.

5.1 Trie-based solutions

Trie-based lookup solutions store the routing table in the form of a trie structure. The main performance evaluation metric for this solution is the number of memory accesses per lookup. Classic software solutions include binary trie [15], path-compressed tries [16], k-stride multibit trie [17], full expansion/compression [18], LC-trie [19], tree bitmap [20], priority trie [21], lulea [22], DIR-24-8 [23], flashtrie [24], shapeGraph [25], and Trie-folding [26].

The binary trie is ideal for updates, but suffers from poor worst-case lookup speed. The Lulea approach requires only about 4 memory accesses per lookup, but can hardly handle incremental updates. DIR-24-8 and full-expansion/compression only need 2 memory accesses per lookup, but the update process is very complicated. Other algorithms strive for a trade-off between lookup speed and update performance, and require multiple memory accesses per lookup.

5.2 TCAM-based solutions

The advantage of TCAMs is that a lookup takes a single memory access, making them very fast. Their main shortcomings are high power consumption, high hardware cost and low density, making them inapplicable to high-end routers. The representative literatures of TCAM-based solutions are [27,28]. The memory access time of TCAMs is close to the one of off-chip memory, which is slower than that the one of on-chip memory.

5.3 Bloom Filter-based Solutions

PBF [4] is the foundational work of Bloom Filter-based solutions. PBF builds one Bloom Filter on-chip and one hash table off-chip, for each level of the trie. Therefore, the matched levels of the trie can be figured out from on-chip membership queries, and the next hop is found through off-chip hash probes. The performance of Bloom filter-based algorithms depends mostly on the query speed of the Bloom filters. Numerous improvements on Bloom Filters have been made to reduce their size or memory access count [5,29]. We refer the reader to [30] for an extensive survey.

To improve on PBF, the authors of [31] propose to use one BF instead of 25 BF, to significantly reduce the computational cost of Bloom Filters. Unfortunately, for each lookup, multiple accesses to off-chip hash tables are necessary. In the best case, one Bloom Filter query and off-chip memory access are needed. However, the worst case is as high as 25 Bloom Filter queries and 25 off-chip memory accesses, which cannot be done in parallel. Note that for IP lookups, the worst case performance is an important metric.

Overall, existing Bloom Filter-based lookup algorithms all require off-chip lookups.

5.4 FPGA-based Solutions

FPGAs [12] have been proposed to accelerate IP lookups. FPGAs are made of on-chip memory (also called block memory) and off-chip memory. The on-chip memory can be organized flexibly, but it is scarce and expensive. The off-chip memory is the ordinary DRAM, which is cheap and large.

The straightforward solution is to store the binary trie in the on-chip memory: each level of the trie is stored in one independent memory stage. Using pipelining, one lookup can be completed in one cycle. The typical work are [32,33]. One main shortcoming of FPGA-based solutions is that the sizes of memory stages differ from one another, while FPGA hardware needs to reserve enough memory in advance. Therefore, Hoang Le *et al.* [34] propose to balance the sizes of different stages at the cost of more complex updates. Another main is-

sue is that on-chip memory cannot hold the binary trie of the current backbone FIBs, and therefore one or more stages have to be stored in the off-chip memory [35]. As a result, the off-chip operations become the bottleneck.

5.5 GPUs and Hashes

GPU-based solutions are able to achieve fast lookup speeds thanks to parallelism across multiple cores [36]. The main challenge of GPUs lies in the data transfer between CPU and GPU. Hash-based solutions pursue few memory accesses at the cost of considerable memory requirements. [37, 38] focus on efficient hash tables with low collision probability, though none of them can hold the hash tables in the on-chip memory and thereby cannot achieve a lookup speed faster than the off-chip memory access speed.

5.6 Splitting algorithms

To lookup MAC tables, Yu *et al.* [39] proposed to split the tables on a per next hop basis. To lookup the FIB, Dharmapurikar *et al.* [4] proposed to split the table into on-chip BFs and off-chip hash tables. Similarly, Song *et al.* [25] proposed to split the FIB into "shape graph" and hash tables. Luo *et al.* [40] proposed to split the FIB into internal nodes and leaf nodes so as to achieve fast updates, requiring both FPGA and TCAM. Similarly, the previous work [41] also splits the FIB in two dimension: 1) prefix length ≤ 24 and prefix length > 24; 2) bit map arrays and next hop arrays. After the split, only a part of the FIB (bit map arrays at level $0 \sim 24$) are stored in the on-chip memory, other parts are stored in the off-chip memory. Different from [41], the goal of this paper is to hold the entire FIB in the on-chip memory.

6. CONCLUSION

Fitting large and ever increasing routing tables in small on-chip memory is like fitting an elephant in a box, which has been considered almost impossible. In this paper, we proposed two *D*imensional *D*ivision *B*loom *F*ilter (D^2BF), a technique able to compactly encode almost all the needed information for performing IP lookup from a FIB in on-chip memory. Furthermore, thanks to pipelining, we showed how to achieve the throughput of one packet per on-chip memory access. We also proposed Sentry Bloom Filters (SeBF) to significantly reduce the false positives that stem from the use of Bloom Filters in D^2BF. We also showed that the probability of accessing off-chip memory to find its next hop is negligible ($1.9 * 10^{-12}$). We simulated our algorithms on FPGA platforms, and through extensive experiments based on publicly available routing data, we showed that we indeed can fit large FIBs in the on-chip memory of today's commodity FPGA platforms, and achieved a throughput of 130Gbps.

7. REFERENCES

[1] X. Meng, Z. Xu, B. Zhang, G. Huston, S. Lu, and L. Zhang, "IPv4 address allocation and the bgp routing table evolution," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 1, pp. 71–80, 2005.

[2] RIPE network coordination centre. [Online]. Available: http://www.ripe.net/data-tools/stats/ris/ris-raw-data

[3] W. Feng and H. Mounir, "Matching the speed gap between sram and dram," in *Proc. IEEE HSPR*, 2008, pp. 104–109.

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

[5] Y. Qiao, T. Li, and S. Chen, "One memory access bloom filters and their generalization," in *Proc. IEEE INFOCOM*, 2011, pp. 1745–1753.

[6] Open source website. [Online]. Available: https://github.com/ddbfcode/DDBFcode

[7] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, 1998, pp. 1–10.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of bloom filters," *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.

[10] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a bloom filter," *Information Processing Letters*, vol. 110, no. 21, pp. 944–949, 2010.

[11] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 5, pp. 651–664, 2010.

[12] FPGA data sheet. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[13] T. Yang, T. Zhang, S. Zhang, and B. Liu, "Constructing optimal non-overlap routing tables," in *Proc. IEEE ICC*, 2012, pp. 2884–2888.

[14] As6447 update report. [Online]. Available: http://bgpupdates.potaroo.net/instability/bgpupd.html

[15] R. Miguel, B. Ernst, and D. Walid, "Survey and taxonomy of IP address lookup algorithms," *Network, IEEE*, vol. 15, no. 2, 2001.

[16] S. Keith, "A tree-based packet routing table for berkeley unix." in *USENIX Winter*, 1991, pp. 93–99.

[17] S. Venkatachary and V. George, "Fast address lookups using controlled prefix expansion," *ACM TOCS*, vol. 17, no. 1, pp. 1–40, 1999.

[18] C. Pierluigi, D. Leandro, and G. Roberto, "IP address lookupmade fast and simple," in *Algorithms-ESA'99*. Springer, 1999, pp. 65–76.

[19] N. Stefan and K. Gunnar, "IP-address lookup using lc-tries," *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 6, pp. 1083–1092, 1999.

[20] E. Will, V. George, and D. Zubin, "Tree bitmap: hardware/software IP lookups with incremental updates," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 97–122, 2004.

[21] L. Hyesook, Y. Changhoon, and S. Earl, "Priority tries for IP address lookup," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 784–794, 2010.

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

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

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

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

[26] 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.

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

[28] 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*, vol. 14, no. 4, pp. 863–875, 2006.

[29] H. Song, F. Hao, M. Kodialam, and T. Lakshman, "IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards," in *Proc. IEEE INFOCOM 2009*, 2009, pp. 2518–2526.

[30] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*. Springer, 2010, pp. 181–218.

[31] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On adding bloom filters to longest prefix matching algorithms," *IEEE Transactions on Computers (TC)*, vol. 63, no. 2, pp. 411–423, 2014.

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

[33] L. Hoang and P. Viktor, "Scalable high throughput and power efficient IP-lookup on fpga," in *Proc. IEEE FCCM*, 2009, pp. 167–174.

[34] H. Le, W. Jiang, and V. K. Prasanna, "A sram-based architecture for trie-based IP lookup using fpga," in *Proc. IEEE FCCM*, 2008, pp. 33–42.

[35] 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*, 2013, pp. 1222–1230.

[36] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010.

[37] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, 2005, pp. 181–192.

[38] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *Proc. IEEE INFOCOM*, 2008, pp. 101–105.

[39] M. Yu, A. Fabrikant, and J. Rexford, "Buffalo: bloom filter forwarding architecture for large organizations," in *Proc. ACM CoNEXT*, 2009.

[40] L. Layong, X. Gaogang, X. Yingke, M. Laurent, and S. Kavé, "A hybrid hardware architecture for high-speed IP lookups and fast route updates," in *Proc. IEEE INFOCOM*, 2012.

[41] T. Yang, G. xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP lookup performance with fib explosion," in *Proc. ACM SIGCOMM, accepted*, 2014.