Towards Accelerating IP Lookups on Commodity PC Routers using Bloom Filter: Proposal of Bloom-Bird

Bahram Bahrambeigy*
Department of Datacenter, Pishgaman Tose Ertebatat (PTE) Inc., Tehran, Iran
b.bahrambeigy@pishgaman.net
Mahmood Ahmadi
Department of Computer Engineering, Razi University, Kermanshah, Iran
m.ahmadi@razi.ac.ir
Mahmood Fazlali
Department of Computer Science, Shahid Beheshti University (SBU), GC, Tehran, Iran
fazlali@sbu.ac.ir

Received: 21/May/2015 Revised: 16/Nov/2016 Accepted: 16/Dec/2016

Abstract

Nowadays, routers are the main backbone of computer networks specifically the Internet. Moreover, the need for high-performance and high-speed routers has become a fundamental issue due to significant growth of information exchange through the Internet and intranets. On the other hand, flexibility and configurability behind the open-source routers has extended their usage via the networks. Furthermore, after assigning the last remaining IPv4 address block in 2011, development and improvement of IPv6-enabled routers especially the open-sources has become one of the first priorities for network programmers and researchers. In IPv6 because of its 128-bits address space compared to 32-bits in IPv4, much more space and time are required to be stored and searched that might cause a speed bottleneck in lookup of routing tables. Therefore, in this paper, Bird as an example of existing open source router which supports both IPv4 and IPv6 addresses is selected and Bloom-Bird (our improved version of Bird) is proposed which uses an extra stage for its IP lookups using Bloom filter to accelerate IP lookup mechanism. Based on the best of our knowledge this is the first application of Bloom filter on Bird software router. Moreover, false positive errors are handled in an acceptable rate because Bloom-Bird scales its Bloom filter capacity. The Bloom-Bird using real-world IP prefixes and huge number of inserted prefixes into its internal FIB (Forwarding Information Base), shows up to 61% and 56% speedup for IPv4 and IPv6 lookups over standard Bird, respectively. Moreover, using manually generated prefix sets in the best case, up to 93% speedup is gained.

Keywords: Bird; Bloom Filter; Forwarding Information Base; IPv4; IPv6; Open Source Routers.

1. Introduction

The need for high-performance and high-speed routers has become a fundamental issue due to significant growth of information exchange through Internet and intranets. Due to adoption of Class-less Inter-domain Routing (CIDR) method, routers need to find best match between various prefix lengths that may differ from lengths 1 to 128 based on what version of IP and what prefix is used. This process of finding matching IPs is time consuming and a lot of hardware (e.g. TCAM and SRAM) and algorithmic approaches (e.g. binary searches) are proposed in the literature as will be discussed further in the related work section.

On the other hand, modern IP router solutions can be classified into three main categories, hardware routers, software routers, and programmable network processors (NPs) [1]. PC-based software routers have created reasonable networking platforms with easy development and programmability features. These features are the most important in comparison with hardware routers. Current software routers reported forwarding up to 40 Gbits/sec traffic on a single commodity personal computer [2]. On the other hand, existence of open-source software routers has brought the opportunity to study and change their codes to make the better routers based on researchers needs. Bird [3], Quagga [4], and Xorp [5] are examples of such open-source routers. Among them, the Bird is selected to implement a Bloom filter (BF) [6] on its internal FIB (Forwarding Information Base) in which all routing tables are based on this data-structure. Since searching in a FIB which stores a huge number of IP prefixes can cause speed bottleneck, we have accelerated it using an extra stage lookup on Bloom filter data-structure. Results show that BF as an extra stage on Bird IP lookups (i.e. Bloom-Bird) makes it up to 93% faster than its standard hashing mechanism for searching big FIBs in the best case. Also results indicate that there is even speedup when result of searching a particular prefix in the FIB is positive because of hash optimizations made in the Bloom-Bird.

The main concern in this paper is to show how a Bloom filter can help an open-source software router to speedup searches when number of inserted nodes and

* Corresponding Author

prefixes becomes huge. In order to have a fair comparison between two versions of Bird (i.e. standard Bird and Bloom-Bird), basic rules and structures of standard Bird is not changed. For example maximum length of Bird's main hash is 16-bits, so it is the same for Bloom-Bird too. The Bloom-Bird includes a Bloom filter array (thus a space overhead) to speedup simple searches for a given IP and length and also Longest Prefix Matching (LPM) lookups. The array can scale its capacity; therefore, False Positive (FP) errors are handled in an acceptable rate.

The main contribution of the paper is proposal of Bloom-Bird router to enhance the performance of Bird open-source router that utilizes a Bloom filter for both IPv4 and IPv6 addresses in its architecture.

The rest of the paper is organized as follows. Section 2 presents related work of Bloom filter and its applications in network processing. Section 3, contains two subsections, which first, is a brief introduction to FIB data structure of Bird and how Bloom-Bird is implemented conceptually, and in the latter subsection the pseudo-codes of implemented approach is presented. Section 4 contains four subsections, which the first two sub-sections are dedicated to IPv4 prefix sets and the last two sub-sections are based on IPv6 prefix sets. Therefore, in Section 4, the first sub-section presents an introduction of the scenario in order to evaluate Bloom-Bird and standard Bird using IPv4 prefixes, and in the second subsection, results of Bloom-Bird evaluation are presented; third and fourth sub-sections are similar to previous sub-sections but they are based on IPv6 prefix sets. Finally, section 5 concludes the paper.

2. Related Work

Bloom filter (BF) is a randomized and probabilistic data-structure proposed by Burton Bloom in the 1970s [6]. BF normally consists of a bit-array representing existence of inserted elements. By checking *k* hash functions and getting negative answer, it can be determined that the element is not inserted certainly. However some FP (False Positive) may occur. Which means BF may express some elements exist by mistake so it needs to check main hash table to make sure about positive answers. Bit-array of BF can reside in an on-chip memory by a hardware implementation to do *k* hash functions checks in parallel. The main advantage of BF structure is Space and Time efficiency in which consumes much less space than ordinary data structures because of its potential collisions and requires much less and more predictable time to query a member.

A lot of variants of BFs are proposed such that more than 20 variants of BF are presented in the literature [7]. Each and every one of them is used for a special manner. For example, standard Bloom filter (SBF) is used in order to check if a specific element is present or not. An important draw-back of SBF is that insertions cannot be undone. Counting Bloom filter (CBF) [8] and later, Deletable BF (DlBF) [9] proposed in order to gain the removability in BF. In CBF each bit in the Bloom array is replaced by a counter. Each insertion, increments counters related to *k* hash functions. Obviously, each deletion decrements related counters. Another variant of BF which supports deletions as mentioned is DlBF. It splits BF array into multiple regions and tracks regions of BF array in which collisions occur. A small fraction of bit-array is used in order to determine related area is collision-free or not. If bits are located in a collision-free region, then the bit can be reset safely, otherwise it will not be safe to delete. Therefore, some bits may not reset if they are located in a collisionary region. CBF is selected for Bloom-Bird because of its simplicity and consistency over deletions instead of DlBF. DlBF would be a good option if BF is going to be implemented in an on-chip memory.

BFs are used in various applications including network processing as discussed in [10] that can be classified into four major categories: Resource routing, Packet routing, Measurement, and Collaborating in overlay and peer-to-peer networks. Moreover, "IP Route lookup" and "Packet Classification" are important applications of BF in the network processing (e.g. [11]). Longest Prefix matching (LPM) or Best Matching Prefix (BMP) that can be classified into IP route lookup category is also an interesting area of BF application. There are a lot of proposed algorithms in order to speedup BMP in the literature. In [12] authors have classified BMP algorithms into "Trie-based algorithms", "Binary search on prefix values", and "Binary search on prefix lengths". "A Trie is a tree-based data-structure allowing organization of prefixes on a digital basis using the bits of prefixes to direct the branching" [12]. Trie-based schemes do a linear search on prefix length because they only compare one bit at a time. The worst case of memory accesses is W when prefix length is W. However, binary search algorithms on prefix values are proportional to log_2N which N is number of prefixes. Binary search on prefix lengths are proportional to log_2W. The first BF application for LPM proposed in [13] which parallel check on on-chip memory is performed to accelerate lookups before checking slower off-chip memory.

Although employing Bloom filters as an extra stage to accelerate IP lookups is well studied by Dharmapurikar et al. [13], Bloom-Bird is different because it is completely independent of specialized hardware implementation and it runs on commodity PC hardware. Therefore, number of hashes is kept as low as possible and the hash probes can be run sequentially and BF can reside in slow memory without loss of efficiency. Moreover, BF on Bloom-Bird helps to accelerate (prefix, length) pair searches in the FIB data structure by ignoring long chains of linked lists of the main hash table. In order to have a fair comparison, basic rules and chain orders of Bird are not modified and modifications are as low as possible.

Furthermore, because IPv6 uses more bits to represent IP addresses (128-bit) compared to IPv4 (32-bit), therefore, it is expected that number of IPv6 prefixes becomes much bigger than current IPv4 prefixes in a near future. Therefore, trie-based schemes will become inefficient. Therefore, multi-bit tries [14] are proposed which compare more than one bit at a time at the cost of space overhead. However, whether Binary search or trie-based lookups is used, it is shown that Bloom filters can accelerate IPv6 lookups too [15], [16]. Nevertheless, the main advantages of our work over the two aforementioned approaches accelerating IPv6 lookups using Bloom filters is that they are based on a special hardware implementation but Bloom-Bird runs completely on commodity PC hardware. To assert again, we didn’t change the main hash of standard Bird into better lookup approaches like trie-based schemes or binary searches, in order to have a fair comparison between standard Bird and Bloom-Bird.

The following section explains Bird open source router fundamental data-structures, pseudo-codes of them and how are they are improved in Bloom-Bird.

3. Bloom-Bird: A Better Bird

“The Bird project aims to develop a fully functional dynamic IP routing daemon primarily targeted on (but not limited to) Linux, FreeBSD and other UNIX-like systems” [3]. It supports latest versions of routing protocols such as BGP, RIP and OSPF. It also supports both IPv4 and IPv6 addresses and a simple command line interface to configure the router. There is a fundamental data-structure called FIB (Forwarding Information Base) in the Bird which routing tables are based on it. This data-structure stores IP prefixes and length of them. Searching in a FIB, where huge number of prefixes is stored can become a speed bottleneck, which can be faster using a CBF (Counting Bloom Filter) as will be presented. Storing in FIB of Bird router is a two stage mechanism. In the first stage, an *order*-bit hash is calculated based on prefix value to find bucket index of main hash table (*order* can be varied from 10 to 16). In the next stage, there is a chain of nodes in linked list structure which may become long due to huge number of nodes. Therefore, a BF can help to reduce of traversing these long chains for missing nodes which results in accelerating the IP lookup mechanism. Nodes are allocated in each chain by Bird Slab Allocator. The implementation of the memory allocator is based on what Bonwick proposed [17] that makes linked list traversing reasonably fast.

To be more specific, there are three important functions related to FIBs in the Bird named fib_get(), fib_find(), fib_route() which are responsible for adding, searching and longest prefix matching, respectively. Each FIB structure in Bird starts with a default 10-bit hash order (i.e. 2^{10}) and increases its hash table size when the number of prefixes increases. This expansion will stop at 16-bit; therefore, chain lengths start to become larger. Implemented BF helps the main hash table when this situation happens to prevent searching in this large FIB chains when an IP prefix cannot be found.

In the following subsection, the way Bloom-Bird implemented is presented conceptually. In the next subsection, the pseudo-codes of implemented approach are discussed.

3.1 Implementation Concepts

Bird uses dynamic hashing size to store prefixes which increases when number of inserted prefixes becomes huge. It starts from 10-bit and it expands until 16-bit order and never grows afterwards. It increments the order by 2 when the capacity limit is reached (e.g. it expands into 12-bit when 10-bit limit is reached). In order to have a fair comparison between standard Bird and Bloom-Bird, these rules are not changed. Therefore, Bloom-Bird includes an extra BF array in each FIB to help it responding faster when it is possible.

Hashing mechanism in the BF of Bloom-Bird is inspired by the main hash table of the standard Bird. If number of entries increases, Bloom-Bird changes size and order of BF array similar to the main hash table approach. Bloom-Bird starts with 18-bit order and it increases to 20-bit order if the capacity limit is reached. This expansion continues until 32-bit and it never grows afterwards. For simplicity, this expansion of BF array is not included in the pseudo-codes in the next subsection. Because of 32-bit order limit, capacity of BF array is limited to 32-bit when an acceptable FP error rate is expected. As the results will show, Bloom-Bird shows at most 15% FP error rate which is fairly good based on Eq. “(1)” in section 5 and tested elements.

In “Fig. 1” a simple Bird’s FIB hashing table is depicted. In the aforementioned Figure, the order can be varied from 10 to 16 as mentioned before. Basic fib_find() function that searches for a given prefix and length in a FIB is shown which uses ipa_hash() function to determine which bucket in main hash table should be used. Main hash array is an *order*-bit array of fib_node type. Afterwards, the node will be inserted into a new free location in the linked lists chain.

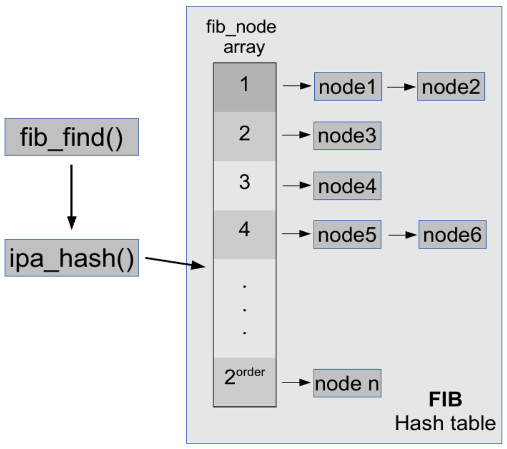

Fig. 1. FIB hashing architecture of standard Bird [3].

In “Fig. 2” the way that BF is implemented in the FIB is depicted. The fib_find() function checks BF for given prefix firstly. If BF confirms the existence of the prefix, then the main hash table will be checked in order to determine pointer address of found node or a FP error may occur. On the other hand, (and more importantly) if BF returns negative answer, checking main hash table will be ignored. Therefore, the main advantage of BF is the latter part in which checking main hash table and maybe traversing long chains of linked lists can be avoided.

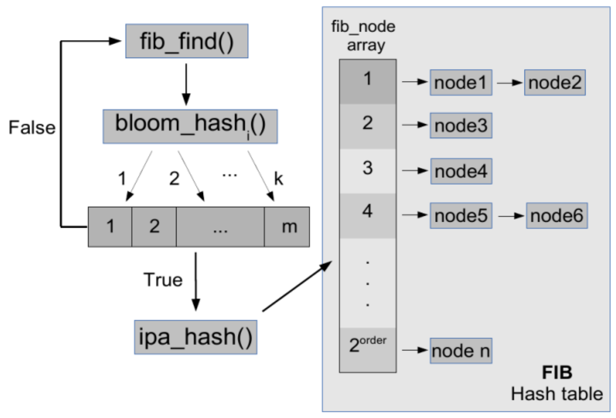

Fig. 2. FIB hashing architecture of Bloom-Bird.

3.2 Implementation Codes

The pseudo-code of fib_find() in the standard Bird (as discussed), is shown as SB_fib_find() function (in order to distinguish between standard Bird and Bloom-Bird functions, SB_ is prepended to Bird functions and BB_ is prepended to Bloom-Bird functions). In the first line, *e* variable points to the selected bucket which is traversed in order to find given prefix. In the second and third lines, the bucket chain is traversed to find the requested node. Two situations may happen after this loop. The loop may find the node, and then last line returns the node. Otherwise, traversing linked list may end with a null pointer; therefore, fourth line returns a null pointer indicating that the node cannot be found.

Psuedo-Code1. SB_fib_find()

```
SB_fib_find(fib, prefix, length)
1. e = fib_table[ipa_hash(prefix)]
2. while((not empty e) AND (not found e))
3.   e = e->next
4. return e
```

In the Bloom-Bird version of this function, in the first line, BF array and its hashing mechanism is used in order to ignore prefixes that do not exist as discussed earlier. The function is changed as BB_fib_find(). Three first lines are dedicated to BF search method for a given prefix. There are *k* hash probes in order to search BF. In each step of the loop, if a location of the BF array represents an empty location then the search returns false answer immediately (i.e. NULL pointer). As it is shown in second line, *k* independent hash functions are used for BF to check the array locations. These hash functions are also depicted in “Fig. 2” *as bloom_hashi().*

Psuedo-Code 2. BB_fib_find()

```
BB_fib_find(fib, prefix, length)
1. for(i=1 to k)
2.   if(filter[bloom_hash_i(prefix)] is empty)
3.     return NULL
4. e = fib_table[hash(prefix)]
5. while((not empty e) AND (not found e))
6.   e = e->next
7. return e
```

Bird uses very simple longest prefix matching (LPM) mechanism that starts from a given length and decrements it until longest prefix match is found or returns a NULL pointer. The pseudo-code is shown as SB_fib_route() function.

Since fib_route() uses fib_find() as its main function to determine existence of the prefixes, BF can help fib_route() very effectively because BF causes no false negative errors and it does not need to go through the main hash chains for lengths that cannot be found. Therefore, there is no need to change anything in the fib_route() function and BF helps LPM indirectly. Although there are better solutions like binary searches on prefix values and lengths as mentioned in related work section, Bird’s LPM algorithm is not changed in order to show BF performance over standard Bird.

Psuedo-Code 3. SB_fib_route()

```
SB_fib_route(fib, prefix, length)
1. while (length ≥ 0)
2.   if(fib_find(fib, prefix, length))
3.     return found node
4.   else
5.     length = length - 1
6. return NULL
```

The last important function is fib_get() which searches for given prefix and length and if does not exist, it adds the prefix into the FIB. The pseudo-code of this function is presented as SB_fib_get() function. It is shown in the first line that the fib_get() uses fib_find() function as its searching mechanism. If it finds the node then the found node pointer will be returned. Otherwise the node will be inserted into selected bucket of the hash table.

Psuedo-Code 4. SB_fib_get()

```
SB_fib_get(fib, prefix, length)
1. if(fib_find(fib, prefix, length))
2.   return found node
3. else
4.   Go down through hash chains
5.   And add new node
```

To check existence of nodes in this function also BF can help through fib_find() when a node does not exist. Therefore, in the first line, BF is checked before its main hash table and when a node is not inserted before, BF counters should be incremented (because CBF is used). Therefore, only one change is needed in this function. This function is been shown as BB_fib_get() function.

Psuedo-Code 5. BB_fib_get()

```
BB_fib_get(fib, prefix, length)
1. if(fib_find(fib, prefix, length))
2.   return found node
3. else
4.   Go down through hash chains
5.   And add new node
6.   for(i=1 to k)
7.     filter[bloom_hash_i(prefix)] += 1
```

Extra lines 6 and 7 of BB_fib_get() function are responsible for updating BF array due to newly added node. This does not count as overhead since hash functions are optimized using bit-wise operations and simplifications compared to standard hash of Bird.

There are two different types of hash functions used in the Bloom-Bird. First type is much like Bird's original hash function which returns a 16-bit hash based on prefix value but optimized using bit-wise operations (ipa_hash() function). Second type hash functions are used for BF which has much less collisions than Bird's original hash function. These second type hash functions return a variety of bit sizes based on BF array length. Number of BF hash functions (*k*) is set to the lowest possible value. Two possible minimum values of k has been tested (i.e. *k=3* and *k=2*). In which *k=3* tests showed a little speed overhead compared to *k=2* tests while the FP error rate was almost the same. Therefore, the *k=2* value is selected for Bloom-Bird to compare its performance with standard Bird.

Therefore, in the Bloom-Bird, *k* is constant and is set to 2 because the loop of checking *k* hash functions becomes speed bottleneck for bigger *k*.

In next Section the scenario and prefix sets in order to compare Bloom-Bird and standard Bird and evaluation are presented and discussed.

4. Evaluation of Bloom-Bird and Results

4.1 IPv4 Scenario

In order to evaluate standard Bird and Bloom-Bird three real IPv4 prefix sets from [18] are gathered from years 2008, 2010 and 2013 sorted by date which latest and more updated one contains more than 482 thousands unique IPv4 prefixes as "Table 1" shows.

Table 1. IPv4 Prefix sets to test the two versions of Bird.

Prefix set alias	# of nodes
Prefix1	262,039
Prefix2	351,645
Prefix3	482,500
Prefix4	1,179,648
Prefix5	1,310,720
Prefix6	2,490,368

The two versions of Bird i.e. standard Bird and Bloom-Bird are evaluated by inserting these real prefix sets and querying them. Prefix sets 4 and 5 are manually generated which contains all possible 24 length prefixes starting with 1-19 and 20-39 octets respectively. Prefix set 6 is concatenation of two prefix sets 4 and 5 in order to test searching FIBs with even bigger prefix sets and make sure about the results. These last three prefix sets contain 99% missing (not existing) prefixes compared to the other three real prefixes (i.e. prefix sets 1-3) in order to show performance of BF when most queries return negative answer (best case). These last three prefix sets are not real prefix traces; therefore, they are only used for searching, not for inserting into FIBs.

Percentage of number of missing nodes when each prefix set is searched is presented in "Table 2". For example when all prefixes in prefix set 2 are inserted into a FIB and all prefixes in the prefix set 1 are queried afterwards, 24.53% of searches return negative answer.

Table 2. Percentage of missing nodes when searching for IPv4 prefix sets

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	0	24.53%	36.27%
Prefix2	43.65%	0	22.92%
Prefix3	65.34%	43.82%	0
Prefix4	99.8%	99.77%	99.56%
Prefix5	99.86%	99.77%	99.39%
Prefix6	99.83%	99.77%	99.47%

There are 0 values in the above Table because the same prefix set is inserted and searched. Results of evaluation are included and discussed in the following subsection.

4.2 IPv4 Results of Bloom-Bird and Discussion

As discussed in the previous subsection, three prefix sets 1-3 are inserted into a FIB at three different times in two versions of standard Bird and Bloom-Bird and all prefix sets 1-6 are queried afterwards. Percentages of speedups of Bloom-Bird (fib_find() and fib_route() functions) over standard Bird and FP error rate are presented in "Tables 3, 4 and 5" respectively. These results are gained on a home PC with 2.88 MHz dual core CPU, 6 MB cache and 4 GB RAM which runs unmodified (vanilla) Linux kernel 3.12.

Table 3. IPv4 Speedups of Bloom-Bird fib_find() over standard Bird - Simple Search Function (*) means the same prefix set is inserted

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	(*) 20%	45%	55%
Prefix2	53%	(*) 17%	47%
Prefix3	61%	58%	(*) 28%
Prefix4	81%	93%	91%
Prefix5	82%	91%	91%
Prefix6	81%	93%	90%

Table 4. IPv4 Speedups Bloom-Bird of fib_route() over standard Bird - LPM Search Function (*) means the same prefix set is inserted

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	(*) 14%	33%	41%
Prefix2	26%	(*) 26%	36%
Prefix3	33%	43%	(*) 32%
Prefix4	41%	62%	64%
Prefix5	44%	63%	63%
Prefix6	42%	63%	64%

Table 5. IPv4 False Positive Percentage of Bloom-Bird
(*) means the same prefix set is inserted

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	(*) 0	1.04%	2.14%
Prefix2	5.46%	(*) 0	1.41%
Prefix3	7.58%	1.25%	(*) 0
Prefix4	8.69%	0.86%	1.66%
Prefix5	9.49%	1.07%	2.3%
Prefix6	9.11%	0.97%	1.88%

In the “Table 3”, speedups of fib_find() function which is responsible for simple searching for a given prefix and length is presented. In the “Table 4”, speedups of fib_route() function which is responsible for Longest Prefix Matching (LPM) is presented (starting length for LPM is set to 32 for all searches). In the “Table 5”, percentage of FP error rate is presented. Also there are 6 rows in the all aforementioned tables, representing what prefix set is searched. The smallest speedup is 14% and biggest speedup is 93%. Smaller speedups are gained when most prefixes are found after search (i.e. number of existing nodes are bigger than missing nodes). On the other hand, bigger speedups are gained when most prefixes are not found after search.

It is well known that FP error can be estimated using following equation [13]:

$$fpr = \left[1 - (1 - \frac{1}{m})^{k*n}\right]^k \quad (1)$$

In which k, m and n represent number of hash functions, size of array, and number of inserted elements, respectively. It gives a nearly accurate estimation and it is used in this paper to evaluate resulted FP errors. The optimal number of hashes (k) can be calculated using following equation:

$$k = \frac{m}{n} \ln 2 \quad (2)$$

Although Eq. “(2)” gives us optimal number of hashes (k) but we need to keep it at its lowest possible value in the Bloom-Bird. Because the higher the k value becomes, the more overhead is caused. That is because of sequential execution of BF array probes in the Bloom-Bird. Therefore, as mentioned before, the number of hashes (k) is set constant number equal to 2.

In order to guarantee its FP rate and performance, the Bloom-Bird calculates its BF array size based on following equation:

$$m = 2^{(n+2)} \quad (3)$$

Therefore, for its default 18-bits order (maximum number of inserted elements can be up to $n=2^{18}$), size of BF array can be calculated based on Eq. “(3)” which leads to $m=2^{20}$. Consequently, given these values of m,n and $k=2$, Eq. “(1)” results 15% FP error rate. Experimental results also show the expected value even in lower rates; As “Table 5” shows, the most FP error rate is 9.49 percent. Therefore, Bloom-Bird handles its FP error rate even better than expected.

The practical FP rate of Bloom-Bird is calculated based on the following equation:

$$fpr = \frac{Observed\ false\ positives}{Tested\ prefixes} \quad (4)$$

Based on the experiments, for small number of prefixes, BF counts only as a memory overhead on Bird i.e. no valueable speedup will be gained. Therefore, BF feature of Bloom-Bird will remain deactivated until its main hash table reaches into 16-bit order. Afterwards, BF array will be allocated and initialized to zero. Hashing mechanism in the BF of Bloom-Bird is inspired by the main hash table of Bird as mentioned before. If number of enteries increases, Bloom-Bird changes size and order of BF feature like the way main hash table does. Bloom-Bird starts with 18-bit order and it increases to 20-bit if the capacity limit is reached (i.e. number of inserted elements reaches $n=2^{18}$). This expansion continues until 32-bit.

In the three “Tables 3, 4, and 5” results show how scaling feature helps accelerating the Bloom-Bird when prefix set 2 is inserted in comparison when prefix set 1 is inserted. Since number of prefixes in the prefix set 2 is bigger than Bloom-Bird default hash order (i.e. 18-bits), the order of BF is scaled up to 20-bits and consequently the FP is decreased in comparison when prefix set 1 is inserted. This situation also shows how FP error rate is important and can make the searches faster.

When “Tables 3 and 4” are compared, the speedups of fib_route() function are lower than its similar situation in the fib_find(). That is because of fib_find() tires just once for given prefix and length but fib_route() tries *W(n)* times in worst case which *n* is 32 for IPv4. Therefore, fib_route() in most cases finds the best match.

For memory usage, number of bits in the BF array can be calculated using Eq. “(3)” as mentioned before. Although 4-bit counters are generally used for counters in CBF, in the Bloom-Bird FIBs, 8-bit counters are used in the CBF because of simplicity and lower overhead of increment operations in the PC for Byte data-type. Since k is constant and is set to 2 and maximum number of inputs by default is 18-bits (i.e. $n=2^{18}$); therefore, the memory requirement for Bloom-Bird in the 18-bits order based on Eq. “(3)” is 1 MB. When the capacity limit is reached, it will be incremented by 2; therefore, it will be 20-bits order. This order requires 4 MB of memory. This expansion continues until 32-bit and the memory requirement can be calculated using Eq. “(3)”.

Similar to two previous sub-sections which IPv4 scenario and results discussed, in the following two sub-sections, the same approach is used but IPv6 prefix sets are used. In the first subsection, scenario is discussed and in the next subsection, results are discussed.

4.3 IPv6 Scenario

Compared to IPv4, unfortunately, latest traces from RouteViews [19] show that existing IPv6 prefixes are very fewer. For example number of latest IPv6 unique prefixes were 16,500 compared to IPv4 which were more than 480,000 unique prefiex. Therefore, in order to show BF advantage of Bloom-Bird over standard Bird, we had to increase the IPv6 prefix sets. For this purpose, ipv6gen [20] tool is used in order to increase number of unique prefixes by calculating possible subnets from exitsing real prefixes. Moreover, just like previous scenario in the first sub-section, three completely manually generated prefixes (not real) are generated using ipv6gen in order to show BF efficiency. The IPv6 prefix sets are shown in the "Table 6".

It should be noted that Bird router converts all IPv6 prefixes into a single 32-bits IP structure using bit-wise OR. It means all previous IPv4 functions can be applied to IPv6 prefixes. The 128-bits prefixes are split into four 32-bits and they are bit-wise ORed into a single 32-bits prefix. Therefore, Bloom-Bird functions can be applied easily to the IPv6 prefixes.

Table 6. IPv6 Prefix sets to test the two versions of Bird.

Prefix set alias	# of nodes
Prefix1	491,136
Prefix2	762,816
Prefix3	1,042,176
Prefix4	2,103,152
Prefix5	2,109,152
Prefix6	4,212,304

Prefix sets 1-3 are based on real IP6 prefix sets gathered from RouteViews [19] from years 2011, 2012 and 2013 sorted by date respectively. The two versions of Bird i.e. standard Bird and Bloom-Bird are evaluated by inserting these real prefix sets and querying them. Prefix sets 4 and 5 are manually generated using ipv6gen [20] tool. Prefix set 6 is concatenation of two prefix sets 4 and 5 in order to test searching FIBs with even bigger prefix sets and make sure about the results. These last three prefix sets contain 99% missing (not existing) prefixes compared to the other three real prefixes (i.e. prefix sets 1-3) in order to show performance of BF when most queries return negative answer. These last three prefix sets are not real prefix traces; therefore, they are only used for searching, not for inserting into FIBs.

Percentage of number of missing nodes when each prefix set is searched, is presented in "Table 7". For example when all prefixes in prefix set 2 are inserted into a FIB and all prefixes in the prefix set 1 are queried afterwards, 12.03% of searches return negative answer.

Table 7. Percentage of missing nodes when searching for prefix sets

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	0	12.03%	19.3%
Prefix2	43.36%	0	10.56%
Prefix3	61.9%	36.83%	0
Prefix4	99.74%	99.72%	99.76%
Prefix5	99.54%	99.47%	99.5%
Prefix6	99.64%	99.6%	99.63%

Results of evaluation based on IPv6 prefix sets are included and discussed in the following subsection.

4.4 IPv6 Results of Bloom-Bird and Discussion

As discussed in the previous subsection, three IPv6 prefix sets 1-3 are inserted into a FIB at three different times in the two versions of standard Bird and Bloom-Bird and all prefix sets 1-6 are queried afterwards. Percentages of speedups of Bloom-Bird (fib_find() and fib_route() functions) over standard Bird and FP error rate are presented in "Tables 8, 9 and 10", respectively. As mentioned in the second sub-section, these results are gained on a home PC with 2.88 MHz dual core CPU, 6 MB cache and 4 GB RAM which runs unmodified (vanilla) Linux kernel 3.12.

Table 8. IPv6 Speedups of Bloom-Bird fib_find() over standard Bird - Simple Search Function (*) means the same prefix set is inserted

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	(*) 8%	30%	33%
Prefix2	49%	(*) 19%	32%
Prefix3	56%	46%	(*) 18%
Prefix4	90%	91%	90%
Prefix5	70%	67%	60%
Prefix6	83%	80%	79%

Table 9. IPv6 Speedups Bloom-Bird of fib_route() over standard Bird - LPM Search Function (*) means the same prefix set is inserted

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	(*) 10%	18%	22%
Prefix2	18%	(*) 14%	22%
Prefix3	20%	24%	(*) 17%
Prefix4	22%	63%	64%
Prefix5	54%	60%	62%
Prefix6	31%	62%	63%

Table 10. IPv6 False Positive Percentage of Bloom-Bird (*) means the same prefix set is inserted

Inserted prefix set / Searched prefix set	Prefix1	Prefix2	Prefix3
Prefix1	(*) 0	2.82%	5.58%
Prefix2	5.85%	(*) 0	3.81%
Prefix3	7.27%	6.5%	(*) 0
Prefix4	4.33%	8.21%	13.18%
Prefix5	3.67%	7.39%	12.07%
Prefix6	4.00%	7.8%	12.62%

In the "Table 8", speedups of fib_find() function which is responsible for simple searching for a given prefix and length is presented. In the "Table 9", speedups of fib_route() function which is responsible for Longest Prefix Matching (LPM) is presented (starting length for LPM is set to 128 for all searches). In the last "Table 10", percentage of FP error rate is presented. Also there are 6 rows in the aforementioned tables, representing what prefix set is searched. The smallest speedup is 8% and biggest speedup is 91%. Smaller speedups are gained when most prefixes are found after search (i.e. number of existing nodes are bigger than missing nodes). On the other hand, bigger speedups are gained when most prefixes are not found after search.

As mentioned before, in order to guarantee FP rate and performance, the Bloom-Bird calculates its BF array size based on Eq. "(3)". Therefore, for its default 18-bits order (maximum number of inserted elements can be up to $n=2^{18}$), size of BF array can be calculated based on Eq. "(3)" which leads to $m=2^{20}$. Consequently, given these values of m,n and $k=2$, Eq. "(1)" results 15% FP error rate. Experimental results also prove the resulted value even in lower values which "Table 10" shows the most FP error rate resulted is 13.18 percent. Therefore, Bloom-Bird handles its FP error rate even better than expected.

When “Tables 8 and 9” are compared, the speedups of fib_route() function are lower than their similar situation in the fib_find(). That’s because of fib_find() tires just once for given prefix and length but fib_route() tries *W(n)* times in worst case which *n* is 128 for IPv6. Consequently, fib_route() in most cases finds the best match.

Although comparing IPv6 scenario to IPv4 scenario is not fair in general because of different number of prefixes and length distribution of them, but speedup of IPv6 compared to IPv4 is a little lower and False Positive errors are a little higher. The only reason for that can be simple hashes that cannot distribute the IPv6 prefixes as well as IPv4 prefixes that use fewer bits to represent prefixes. Therefore, False Positive errors because of simple IPv6 hashes become bigger and speedups become lower compared to IPv4 scenario.

5. Conclusion

The paper showed and presented another application of Bloom filter on a practical open-source router. The BF implementation on Bird’s FIB data structure showed that it can help Bird to search and route faster when number of inserted prefixes into a FIB becomes huge. Bloom-Bird which utilizes a Bloom filter in its architecture, evaluated using various prefix sets gathered from real routers traces and also manually generated prefix sets to make the tests more accurate and reliable. Bloom-Bird employs a Bloom-filter in Bird’s FIB data structure in order to accelerate the IP lookups when FIB’s linked list chains become long. Comparison using different prefix sets showed that up to 93% speedup is gained when most searches return negative answer. This improvement is achieved at the cost of Bloom filter space overhead. Moreover, it is showed how Bloom-Bird can handle its FP error rate when number of inserted prefixes increases by scaling the Bloom filter capacity. The results presented and discussed for both IPv4 and IPv6 prefix sets.

Regardless whether Bloom filter is going to be used as an extra stage before hashing mechanism or other searching data structures (e.g. trie), it can help to avoid traversing chains and paths when result of a search is negative. Therefore, our software based approach is applicable to any other software based routers to accelerate their IP lookups when their FIBs become huge.

References

[1] Y. Zhu, Y. Deng, and Y. Chen. "Hermes: an integrated CPU/GPU microarchitecture for IP routing," presented at the 48th Conf. Design Automation Conference, San Diego, California, 2011.

[2] S. Han, K. Jang, K. Park, and S. Moon. "PacketShader: a GPU-accelerated software router," ACM SIGCOMM Computer Communication Review, vol. 41, pp. 195-206, 2010.

[3] O. Filip. “The BIRD Internet Routing Daemon Project” Internet: www.bird.network.cz/?index, Jun. 15, 2013 [Mar. 7, 2017].

[4] P. Jakma. “Quagga Software Routing Suite.” Internet: www.nongnu.org/quagga, Dec. 6, 2015 [Mar. 7, 2017].

[5] M. Handley, O. Hodson, and E. Kohler. "XORP: an open platform for network research," ACM SIGCOMM Computer Communication Review, vol. 33, pp. 53-57, 2003.

[6] B. Bloom. "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, vol. 13, pp. 422-426, 1970.

[7] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. "Theory and practice of bloom filters for distributed systems," Communications Surveys & Tutorials, IEEE, vol. 14, pp. 131-155, 2012.

[8] L. Fan, P. Cao, J. Almeida, and A. Broder. "Summary cache: a scalable wide-area web cache sharing protocol," IEEE/ACM Transactions on Networking (TON), vol. 8, pp. 281-293, 2000.

[9] C. Rothenberg, C. Macapuna, F. Verdi, and M. Magalhães. "The deletable bloom filter: a new member of the bloom family," IEEE Communications Letters, vol. 14, pp. 557-559, 2010.

[10] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey," Internet Mathematics, vol. 1, pp. 636-646, 2002.

[11] M. Ahmadi and S. Wong. "Modified collision packet classification using counting Bloom filter in tuple space," presented at the 25th Int. Multi-Conference: Parallel and distributed computing and networks, Innsbruck, Austria, 2007.

[12] L. Hyesook and L. Nara. "Survey and proposal on binary search algorithms for longest prefix match," Communications Surveys & Tutorials, IEEE, vol. 14, pp. 681-697, 2012.

[13] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. "Longest prefix matching using bloom filters," presented at the Conf. on Applications, technologies, architectures, and protocols for computer communications, Karlsruhe, Germany, 2003.

[14] S. Sahni and K. S. Kim. "Efficient construction of multibit tries for IP lookup," IEEE/ACM Transactions on Networking, vol. 11, pp. 650-662, 2003.

[15] K. Lim, K. Park, and H. Lim. "Binary search on levels using a Bloom filter for IPv6 address lookup," presented at the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Princeton, New Jersey, 2009.

[16] S. Haoyu, H. Fang, M. Kodialam, and T. V. Lakshman. "IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," in INFOCOM 2009, IEEE, 2009, pp. 2518-2526.

[17] J. Bonwick. "The slab allocator: an object-caching kernel memory allocator," presented at the Technical Conf. on USENIX Summer 1994, Boston, Massachusetts, 1994.

[18] D. Meyer. “RouteViews IPv4 BGP RIBs. 2008, 2010, 2013.”, Internet: www.routeviews.org/bgpdata Feb. 28, 2017 [Mar. 7, 2017].

[19] D. Meyer. “RouteViews IPv6 BGP RIBs. 2011, 2012, 2013.”, Internet: www.routeviews.org/bgpdata Feb. 28, 2017 [Mar. 7, 2017].

[20] V. Kotal, “IPv6 prefix generator.”, Internet: www.github.com/vladak/ipv6gen Jan. 29, 2011 [Mar. 7, 2017].

Bahram Bahrambeigy received his B.Sc degree in Information Technology Engineering and M.Sc degree in Computer Networks Engineering both from Islamic Azad University (IAU) in 2011 and 2013, respectively. He is currently working at Pishgaman Tosee Ertebatat (PTE) Tehran as Datacenter Engineer. His research interests include Computer Networks, Software Routers and High-performance Computing.

Mahmood Ahmadi received the B.Sc degree in Computer engineering from Isfahan University, Isfahan, Iran in 1995. He received the M.Sc degrees in Computer architecture and engineering from Tehran Polytechnique University, Tehran, Iran in 1998. From 1999 to 2005, he was a faculty member at Razi university in Kermanshah in Iran. In October 2005, he joined the Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS), Delft University of Technology, Delft, The Netherlands, as fulltime Ph.D student. He got his Ph.D in May 2010. His research interests include Computer architecture, network processing, Bloom filters, software defined networking, and high-performance computing. He is working as an assistant professor at Computer Engineering Department in Razi University of Kermanshah, Iran.

Mahmood Fazlali Mahmood Fazlali received B.Sc in computer engineering from Shahid Beheshti University (SBU) in 2001. Then he received M.Sc from University of Isfahan in 2004, and Ph.D from SBU in 2010 in computer architecture. He performed researches on reconfigurable computing systems in computer engineering lab of Technical University of Delft (TUDelft) as a postdoc researcher. Now, he is working as an assistant professor at computer science department at SBU. His research interest includes high performance computing, parallel processing, reconfigurable computing and computer aided design.