Towards practical use of Bloom Filter based IP lookup in operational network

Tong Yang, Gaogang Xie, Ruian Duan, Xianda Sun, Kavé Salamatian

Towards Practical Use of Bloom Filter based IP Lookup in Operational Network

Tong Yang, Gaogang Xie
Institute of Computing Technology,
Chinese Academy of Sciences (CAS), China.
yangtongemail@gmail.com, xie@ict.ac.cn.

Ruian Duan
College of Computing,
Georgia Institute of Technology, USA.
ruian@gatech.edu.

Xianda Sun
David R. Cheriton School of Computer Science,
University of Waterloo, Canada
x6sun@uwaterloo.ca

Kavé Salamatian
University of Savoie, France.
kave.salamatian@univ-savoie.fr

Abstract—**Bloom Filter is a widely used data structure in computer science. It enables memory efficient and fast set membership queries. Bloom filter-based solutions have been proposed in the past decade for lookup in forwarding tables of backbone routers [2]. However, the main shortcomings of using Bloom Filters for lookup lie in the absence of support for deletion operations that are needed to update the forwarding tables. Counting Bloom Filter supporting deletion has therefore to be used, increasing significantly the memory requirement. Moreover, Counting Bloom Filter suffers from both false positive and false negative. In this paper, we propose to solve the issue with deletion of Bloom Filters by using a Withdrawal To annOuncement (WTO) mapping that replaces withdrawal with announcements, transforming deletions into additions or record changes. Moreover, as nowadays routing updates are becoming more and more frequent, and back routers routing tables are inflating, the number of entries in a Bloom filters increases, increasing the risk to saturate them, especially during bursty updates. In order to limit the worst case false positive rate, the size of Bloom filters should be dynamicly adjusted. We address this issue by an algorithm to Dynamicly Increase the Size of Bloom Filters (DISBF). Experimental evaluation show that the proposed techniques improve largely the performance of Bloom Filter used for forwarding lookup and open way for the use of Bloom Filters in real operational settings.**

I. INTRODUCTION

Bloom Filter (BF) data structures have been applied to a large set of applications in computer science [5][6][7]. Bloom Filters are used for fast and memory efficient set membership queries. In the past decade, applications of these structures to networking problems have been proposed.

In order to increase lookup speed and reduce its cost and power consumption, Dharmapurikar *et al.* [2] proposed the Prefix Bloom Filter (PBF**)** structure that uses on-chip Bloom Filter to represent the trie[1] used to find the longest matched prefix. The evaluation shows that in average about 1.003 off-chip memory accesses is needed for any single lookup, faster than TCAM in average but with larger worst case complexity. PBF only uses SRAM and achieved therefore lower cost and lower power consumption. However, any lookup solution has to deal with updates that are frequent in the current operational network. Unfortunately, Bloom Filter cannot support deletion operations that are needed to do updates. Therefore, they have to be replaced with Counting Bloom Filter (CBF) that uses a *k* bits counter to replace each bit of Bloom Filter array, *i.e.*, supporting deletion operations CBFes entails *k* times more memory. This can prevent CBFs to be stored in on-chip FPGA memory. Moreover, in addition to false positive that is common in BFs, CBF can suffer from false negatives happening when a counter overflows [4]. False negative results in wrong lookups that are not acceptable for ISPs.

Nonetheless, while deletion operations are problematic for Bloom Filters, insertion operations are natural. We propose in this paper the Withdrawal To annOuncement (WTO) mapping that transforms withdrawal messages to announcement messages, that have the same effect on the forwarding behaviour of the routing table. The technique is motivated by the fact that when a prefix is withdrawn in a forwarding or routing table, it always has a shorter less specific prefix that has a default next-hop. The idea of WTO mapping is therefore to transform a prefix deletion message into a prefix insertion (or change) with the next-hop set to the next-hop of the closest ancestor prefix node. We present the details of WTO mapping in Section IV. As most of update and withdrawals happen in the leaves, WTO mapping achieves excellent performances.

During the past decade, backbone routers have witnessed a steady growth of their routing tables size. This implies Bloom Filters that should be used for forwarding lookup have to integrate an increasing number of entries. This results in increased false positive rate and eventual saturation of the Bloom Filter (when most bit in the bitmap are set). This increasing false positive rate means more hash probes in off-chip memory for PBF algorithm, and ultimately degrades the system performance. In order to limit the worst case false alarm rate, the on-chip BFs should be dynamicly adjusted with the number of elements stored in the BF. Several approaches have been proposed to deal with this. Yu *et al.* [8] proposed to partition the forwarding table according to the outgoing port (outgoing link), and to build a specific BF for each outgoing

[1] Trie is a tree-like data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching, an excellent survey of trie-based lookup solutions are provided in [1].

port. This BF is periodically reconstructed with the aid of a CBF and its size is increased accordingly. However, during the process of reconstructing BFs, packet forwarding should be suspended and arriving packets are inserted into a queue. This increases the risk of packet dropping and this is highly undesirable for ISPs. To address this issue, we propose a scheme that can dynamicly increase the size of BF, named DISBF algorithm. The core idea of DISBF algorithm is to assign additional memory to ensure that the false positive does not increase during the insertion updates. The cost of this dynamic adaption is a few additional memory accesses. The details are provided in Section IV.

The rest of the paper is organized as follows. Section II introduces the background, including IP lookup, forwarding table, and the principle of Bloom Filters. Section III details the WTO mapping and Section IV describes the DISBF algorithm. Performance evaluation is provided in Section V, and finally we conclude our paper in Section VI.

II. BACKGROUND

A. Bloom Filter Theory

Bloom Filter is a space–efficient data structure used for set membership queries. A Bloom Filter has two main components an *m* bits array, and *k* hash functions. When a new entry is to be added to the BF, it is hashed by each one of the *k* hashes to a value from *1* to *m* and the corresponding value is set to in the *m* bits vector.

1) Initialization

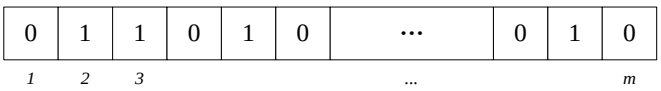


Figure 1. The initial state of Bloom Filter.

In the beginning, as shown in Figure 1, the Bloom Filter is an m-bit array, every bit of which is 0. There are *k* mutually independent hash functions to map every element to *k* positions with a uniform random distribution.

2) Insertion

As shown in Figure 2, given a set $X = \{x_1, x_2, x_3, \dots, x_m\}$, for each element x_i in X, each hash function will compute an array position, which will be set to 1. Therefore, each element causes *k* bits in the m-bit array to be set to 1. Note that if the position of a hash value is already set to 1, no change is needed for standard Bloom Filter[2].

After the insertion of all the elements of the set X, the construction of Bloom Filter is complete, and then membership query[3] can be performed.

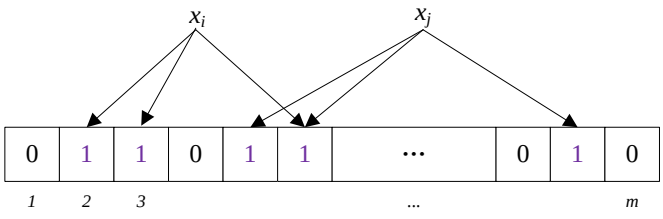


Figure 2. The insertion of Bloom Filter.

[2] For Counting Bloom Filter, the position will increase by one.

[3] For convenience, the process of determining whether or not an element is a member of a set is called membership query in this paper.

3) Membership Query

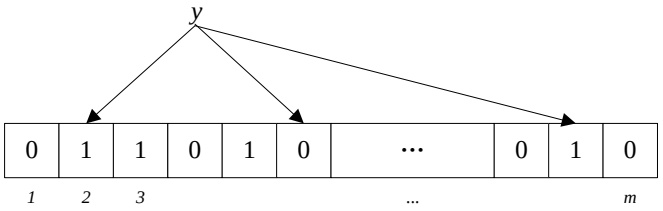


Figure 3. Membership query of Bloom Filter.

As shown in Figure 3, given an element *y*, the same *k* hash functions compute *k* positions in the array. If all the *k* bits corresponding to the *k* hash values are 1, the element is judged to be an element of the set X with a probability of false judgment; otherwise, it is definitely not a member of the set X.

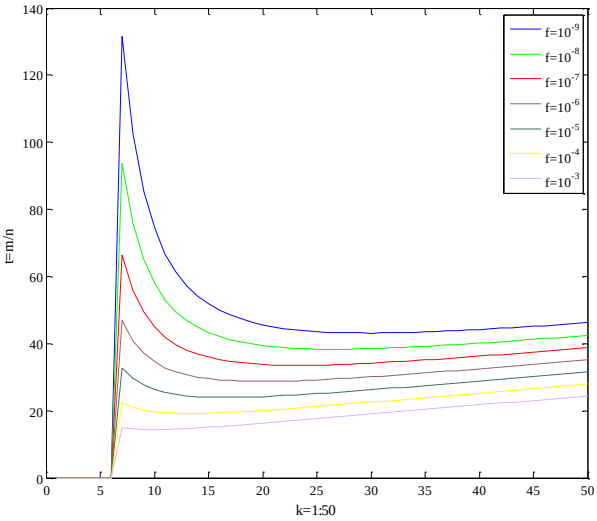


Figure 4. The relationship between *t* and *k* given the value of *f*. Given a set with *n* elements, the memory occupation is *tn* bits. We need to choose the optimal *k* and *t* to achieve the expected possibility of false positive.

However, even if an element *z* is indeed not a member of the set X, the *k* hash positions might be all 1, this is called false positive. The probability of false positive has been well researched [3][4], thus we only present the important conclusions below:

Suppose *m* is the size of the filter (the length of bit-vector), *k* is the number of hash functions, and *n* is the size of the element set. *f* represents the false positive probability, and it is given by:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k$$

When *m* is large, the above equation can be simplified to:

$$f \approx \left(1 - e^{\frac{-nk}{m}}\right)^k$$

Given the value of *f*, the following relationship can be concluded:

$$m \approx -\frac{k}{ln\left(1 - f^{\frac{1}{k}}\right)}n$$

$$t = \frac{m}{n} = -\frac{k}{ln\left(1 - f^{\frac{1}{k}}\right)}$$

The false positive probability is minimized when satisfying

$$k = \frac{m}{n} ln2 \quad (1)$$

and

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

Given the value of the possibility of false positive, we can compute the optimal values of *m* and *k* according to equation (1) and (2).

To simplify the description of the memory requirement of Bloom Filter, we define *t*=*m*/*n*, *i.e.*, given a set with *n* elements to insert into a Bloom filter, the needed memory is *tn* bits. The optimal *k* and *t* to achieve an expected probability of false positive is given by the above relations. Figure 4 shows the relationship between *t* and *k* when the value of the possibility of false positive is given. For example, if *f* is required to be 10^{-8}, *k* should be 21, and *t* should be 39.05, *i.e.*, *m*=39.05*n*. For another example, if *f* is required to be 10^{-4}, *k* should be 13, and *t* should be 19.17, *i.e.*, *m*=19.17*n*.

B. Bloom Filter applications for IP lookup

The initial paper that proposed to use Bloom Filter for packet forwarding in IP networks proposed Prefix Bloom Filters (**PBF**) lookup algorithm [2]. This algorithm first builds a trie structure for the routing table. It regroups after the prefixes in each level with the same prefix length as a prefix set. PBF consists of two steps: on-chip [4] Bloom Filter query determining the matched trie level (the longest matched prefix length[5]) and off-chip hash probes looking-up the next-hop, *i.e.*, a Bloom Filter is built in the FPGA on-chip memory for each prefix set and an off-chip hash table for each level of the trie. This results in 32 Bloom Filters and 32 hash tables. Given an incoming IP address, we first query its high 1bit, 2bits, 3bits, …, 32bits prefixes in the corresponding 32 Bloom filters. Due to prefixes overlap, several Bloom Filters can report a match. According to the Longest Prefix Matching (LPM) rule, PBF checks the longest matched level in the corresponding off-chip hash table and returns the corresponding egress port. When the longest Bloom Filter returns a false positive, the search in the hash table will not find any corresponding record and PBF has to check the hash table for the second longest prefix. This process continues until a match is found in hash table.

Membership query of Bloom Filter is so fast that it is considered to be able to be ignored compared with off-chip operations, and thereby the off-chip hash probes become its bottleneck. Indeed above-described PBF can achieve fast lookup with low cost, but there are two main shortcomings: updates, and routing table growth.

Nowadays, networks have to deal with an increasing number of update and routing table changes. Therefore, the performance of incremental update has become a very important metric for routing lookup. Solutions that suspend lookup too long during routing table updates are highly undesirable. PBF authors have proposed to use Counting Bloom Filters in place of Bloom Filter to deal with this issue. In CBF, each bit of the membership array of Bloom Filter is replaced by a *k* bits counter that is incremented every time, and the corresponding position is set by the insertion of a new element in the CBF. This enables to delete one element from the CBF by simply decrementing the counters set by the insertion of the deleted element. However, the ability to delete elements from a CBF comes at the cost of *k* fold memory requirements. Therefore, for a given memory footprint the false positive rate of a CBF becomes larger than the false positive rate of a BF. Moreover, the counters of a CBF can overflow (when the number of elements setting a position goes higher than 2^{k-1}). This leads to false negative, *i.e.*, claiming that an element is not in the Bloom Filter when it is in it. The false negative has more serous impact on routing than false positive as when they occur, there are two error cases: 1) The longest matched level is missed and the packet is forwarded to the wrong next-hop; or 2) no Bloom Filter reports match and the next-hop remains unknown. These points make PBF unsuitable for real routers.

A second issue is relative to handling fast inflating routing tables. With routing table keeping a rapid growth during recent years, the on-chip Bloom Filters have to hold an increasing number of prefixes. With fix memory size this incurs an increase of the false positive rate, and a resulting increase of costly off-chip hash lookups, ultimately decreasing the lookup speed. To keep the system throughput one should manage to limit the false positive rate while the number of entries in the Bloom Filter increases by dynamicly adjusting the size of the Bloom Filter. Unfortunately, there has been no solution to dynamicly increase the size of Bloom Filters.

III. WTO ALGORITHM

First we give some conclusions of Bloom Filter and Counting Bloom Filter:

- Bloom Filter supports insertion operations and membership query, but cannot support deletion operations.
- Bloom Filter has false positive, but no false negative.
- Counting Bloom Filter uses a counter in place of a bit of Bloom Filter, hence can support deletion operations.
- Counting Bloom Filter has both false positive and false negative.

In order to support incremental update, PBF adopts Counting Bloom Filter, then two problems arise:

- If a counter in the Counting Bloom Filter costs *x* bits, then *x* times memory is needed as compared with Bloom Filter. Therefore, Counting Bloom Filter is probably too large to be held in on-chip memory.
- Counting Bloom Filter has false negative. When false negative occurs, PBF algorithm probably returns a mistaken next-hop.

Actually, the above two problems can be solved if there is no withdrawal message, but we cannot just ignore the withdrawal messages.

Deletions are needed when withdrawal messages happen. A withdrawal message means that the announced prefix and its

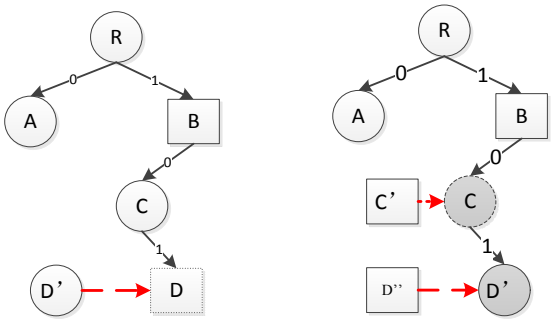
[4] For FPGA [10] and ASIC, there are on-chip and off-chip memory: on-chip memory is fast but scarce, while off-chip memory is slow but abundant.

[5] The routing table can be stored in a binary trie, and the prefixes with the same length will be stored in the same level of the trie.

next-hop should be deleted from routing table, *e.g.*, let's assume that the prefix1011*:4 has to withdraw. Now, if a packet with destination address IP 10110* arrives, to which egress should it be forwarded? In practice, there are always shorter prefixes matching the IP, like 101*, 10* and *, in the routing table. So in this example, the withdrawal message just changes the longest matched prefix from 1011* to 101*. Therefore, in place of deleting the prefix 1011*, one can just change its next-hop to the next-hop of prefix 101*. By doing this the forwarding behavior of the routing table will change. Nonetheless, we have just transformed the withdrawal messages into an announcement messages and suppress the need for a deletion operation in Bloom Filters. This example illustrates the rationale of Withdrawal To annOuncement (WTO) mapping algorithm. WTO algorithm aims at changing withdrawal message into announcement one while keeping the forwarding behavior of routers unchanged.

By eliminating the need for deletion, WTO algorithm enables to simply use a Bloom Filter instead of the Counting Bloom Filter.

The WTO mapping algorithm works as follows. It seeks a way to convert withdrawal messages to announcement messages by changing the next-hop of the prefix to be deleted to its nearest ancestor's next-hop in the trie. For instance, as shown in Figure 5, node A, B, C, D and R are 4 prefix nodes in a trie and the circles represent that the egress port of these nodes is 1, while the rectangles represent port 2. When a withdrawal message: withdraw 101*, arrives instead of removing node D and updating the Bloom Filter, WTO algorithm changes the next-hop of node D to port 1 (the next-hop of node C), suppressing the need to apply an operation on the Bloom Filter.



(a) step I. (b) step II.

Figure 5. The scheme of WTO algorithm. Node A, B, C, D and R are 4 prefix nodes in a trie and the circles represent that the port of such nodes are 1, while the rectangles represent port 2. Given a withdrawal message: withdraw 101*, instead of simply removing node D and updating the Bloom Filter, WTO algorithm changes the next-hop of node D to port 1 (the next-hop of node C), thus no operation on Bloom Filter is needed. When deleting node C after node D is deleted, WTO algorithm needs to change the port of node C to port 2, and the port of node D should be changed as well.

It is noteworthy that WTO algorithm may cause domino effect. In the example shown in Figure 5, when deleting node C after node D being deleted, WTO algorithm needs to change both the egress port of node C and D to port 2. To address this potential problem, a simple solution consists in checking the sub-trie rooted at the updated node. However, this entails a longer time and larger memory. To accelerate WTO mapping, we assign a flag to each node. When during deletion operation, the updated node is not really deleted but changed, just like the node D in step I of Figure 5, the flag of the nearest ancestor node (node C in Figure 5) should be set to true. During next deletion, WTO algorithm finds that the flag of a node is true, it continues to traverse the sub-trie to change the corresponding nodes. This happens in the example of Figure 5, when node C is deleted after node D.

However, getting the nearest ancestor and its flag needs a pointer in each node to point back to its parent node. To avoid this back pointer, we record the flag of the nearest ancestor node in the next node during the traversal process needed to find the updated node. This further reduces the additional memory accesses resulting from back pointers..

The above technique reduces strongly in practice the needed sub-trie traversals. We give in Algorithm 1 the pseudo code of WTO algorithm.

Nonetheless, when the domino effect happens, it needs more time to apply an update than the common situations. The rate of domino effect depends on the update messages. In other words, the performance of WTO algorithm depends on the characteristics of update messages, in particular the withdrawal update messages. According to our previous experimental results in [12], the update messages happen mainly in *Leaves nodes*. This suggests that even when the domino effect happens, only a few levels of the prefixes are affected.

Algorithm 1 Update(operationType, prefix, next-hop)

```
1: insertNode = trie.root
2: next-hopAncestor = insertNode.newPort
3: for {i = 0 to prefix.length-1}
4:         if {prefix[i] == 0} THEN
5:                 if {insertNode.leftChild == NULL} then
6:                         if {operationType == WITHDRAW}
7:                         return;
8:                         endif
9:                         create insertNode.leftChild
10:                endif
11:                next-hopAncestor = insertNode.newPort
12:                insertNode = insertNode.leftChild
13:        else
14:                if {insertNode.rightChild == NULL} then
15:                        if {operationType == WITHDRAW}
16:                        return;
17:                        endif
18:                        create insertNode.rightChild
19:                endif
20:                next-hopAncestor = insertNode.newPort
21:                insertNode = insertNode.rightChild
22:        endif
23: endfor
24: if {operationType == ANNOUNCE} then
24:        insertNode.oldPort = next-hopAncestor = next-hop
26:        [Bloom Filter Operations]
27: else
28:        insertNode.oldPort = 0
29: endif
30:if{flag==true} then
30:        traversalUpdate(insertNode, next-hopAncestor)
```

In addition to this, we have also carried out large-scale experiments and find that withdrawal messages are much fewer

than announcement messages (see Figure 8 in Section V). This phenomenon indicates that WTO algorithm is applicable for real routers. As explained above, the WTO mapping can fix the of the major problems that lead authors of [2] to use CBFs in place of BF, controlling therefore the memory increase. Moreover, the increase in the number of memory accesses induced by domino effect of the WTO algorithm is not too large to result in performance loss (see Figure 10 in Section VI).

IV. Dynamicly Adjusting the Size of Bloom Filters

A. DISBF Algorithm

Routing table size has taken a large increase in recent years, *e.g.*, the Autonomous System 6447 routing table had only about 70K prefixes in 2000, but exceeded 450K prefixes at the beginning of 2013 [14]. One issue that such an explosive increase can generate for Bloom Filter based lookup is that the number of entries to add into the Bloom Filter increases and the false positive rate consequently increases. In order to control the false positive rate we need to adapt the size of the Bloom Filter dynamically.

This issue has already been analysed. Yu et al. proposed in [8] to re-construct periodically the Bloom Filter with the aid of Counting Bloom filters. Unfortunately, re-constructing the Bloom Filters interrupt the lookup of forwarding table, and might incurs packet loss. It is therefore not applicable in practice.

We propose an algorithm to **D**ynamically **I**ncrease the **S**ize of **B**loom **F**ilter (**DISBF**) that do not need to reconstruct the BF and avoid blocking the lookup process. Note that we also aim at keeping the false positive rate almost unchanged with increasing the number of elements inserted into the BF.

Let's assume that in Figure 6, x_i is an element of to add to a Bloom Filter. The original BF membership array is the bits ranging from *0* to *m*, while the memory (the purple part) ranging from *m+1* to *m'* is the additional memory. h_s is one of the *k* hash functions.

Now, let's define Bloom Filter to be 'full' when the number of inserted entries becomes larger than a threshold defined by the maximal acceptable false positive rate. Now let's assume the BF becoming 'full'. The DISBF algorithm consists of 4 steps (see Figure 6):

1) We compute a new size *m'* for the Bloom filter, and 'malloc' additional *m'-m* memory for it.

2) When inserting a new element, we still compute the *k* hash functions, and check the corresponding bit $h_s(x_i)$. If the bit is set (is 1), nothing needs to be done. However, if the value is 0, we do not set it, but compute the $h_s(x_i)\ mod\ (m-m')$, and set the corresponding bit in the additional memory to *1*.

3) When querying an element *x*, check the original *k* hash position: if all 1, report true; otherwise, if position indicated by the hash function *i* is 0, go to step 4).

4) We check the positions $h_i(x)\ mod\ (m-m')$ in the additional memory, if it is 1, we report true, otherwise, if there is a further round of added memory we apply the same approach. If in the last memory region the value is 0, we report a false.

It is noteworthy that the above method does not add false negative, and the false positive rate is not increasing due to the addition of memory. Moreover we do not need any additional hash computations but only additional modulo operations.

In conclusion, our scheme can dynamically increase the size of Bloom Filter, avoiding Bloom Filter reconstruction without adding false negative, and keeping the false positive below limit. No additional hash function and hash computation is needed. The additional modulo operations and corresponding memory accesses are also very rare (happening only for entries added after the Bloom Filter becoming 'full').

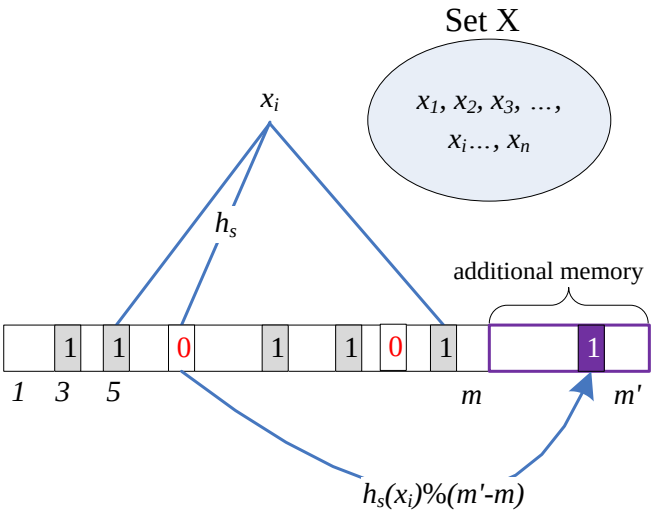


Figure 6. DISBF algorithm. x_i is an element of the set. The memory ranging from *0* to *m* is the original Bloom Filter, while the memory (the purple part) ranging from *m+1* to *m'* is the additional memory. h_s is one of the *k* hash functions.

B. The False Positive of DISBF

As mentioned before, the false positive probability of Bloom Filters stays bounded by using DISBF. In this section, we analyze formally the false positive probability of DISBF.

Let's suppose that before adding additional memory, *n* elements are inserted into the Bloom Filter that is using *k* hash functions, and a membership array of size *m*. When *m* is large, the false positive probability *f* is given by :

$$f \approx \left(1 - e^{\frac{-nk}{m}}\right)^k$$

As explained in section II.A, the optimal value of false positive is

$$f_{opt} = \left(\frac{1}{2}\right)^k$$

and the values *k*,*m* and *n* are related by the below relation:

$$k_{opt} = \frac{m}{n} ln2$$

When *k* is fixed, $\left(1 - e^{\frac{-nk}{m}}\right)^k$ and *f* increase with *n*. We will assume that the Bloom Filter is becoming 'full' when the number of inserted elements exceeds n_{max}, resulting in a false positive probability less than

$$f_{max} \approx \left(1 - e^{\frac{-n_{max}\ k}{m}}\right)^k$$

Now if we keep *k* unchanged, and

$$\frac{m_c}{n_c} = \frac{m}{n}$$

The false positive probability still remains:

$$f_c = \left(\frac{1}{2}\right)^{\frac{m_c}{n_c}ln2} = \left(\frac{1}{2}\right)^{\frac{m}{n}ln2} = f_{opt}$$

When $n_c > n_{max}$, the DISBF algorithm 'malloc' m'-m additional memory. We use f' to represent the probability of false positive using DISBF algorithm. When $n_c = n_{max}$, all the bits in the additional memory are 0, thus

$$f' = f \approx \left(1 - e^{\frac{-nk}{m}}\right)^k$$

When the probability of false positive reaches the predefined threshold f_{max}, DISBF is activated, and the number of *1s* in the original Bloom Filter array is not anymore increased by additional insertion. The rationale of DISBF scheme is to transfer the newly inserted *1s* to additional memory. Now let's suppose that after inserting x additional elements into the Bloom Filter, the *1s* are evenly distributed in the added bits, and

$$\frac{m}{n}ln2 = \frac{m'}{x + n_{max}}ln2 = k_{opt}$$

The false alarm probability f for a bloom filter not using the DISBF algorithm will become:

$$f \approx \left(1 - e^{\frac{-(n_{max}+x)k}{m}}\right)^k$$

Now the DISBF, has larger size and fewer *1s*, compared with BF, thus

$$f' \approx \left(1 - e^{\frac{-(n_{max}+x)k}{m}}\right)^k < \left(1 - e^{\frac{-(n_{max}+x)k}{m}}\right)^k = f$$

The same argument is valid when after adding more entries, the new Bloom Filter becomes 'full', *i.e.*, the probability of false positive reaches the predefined threshold f_{max}. In this case, new additional memory is assigned and the four steps of DISBF algorithm are executed again.

V. Experimental Result

In this section, we will validate using empirical experiment the proposed techniques to implement Bloom Filter based IP lookup.

A. Experimental Settings

1) Data Set

The data set is taken from RIPE NCC [11] at www.ripe.net, which collects routing updates from peers. In order to objectively evaluate the performance of WTO algorithm, we extracted the RIB on 2012/6/1 at 8:00 AM from 10 backbone routers, and all corresponding update messages happening during a full day are downloaded and parsed.

2) Computer Configuration

Our experiments have been conducted on a windows XP sp3 machine with Pentium (R) Dual-Core CPU 5500@2.80GHz and 4G memory.

B. Experiments on WTO Algorithm

The *x*-axis of Figure 7~12 represents the update time of update messages. For instance, '201210231945' means the time 2012-10-10 23:19:45. The labels, rrc00, rrc01, *etc.* are the router ID defined by RIPE Network Coordination Centre [11].

As the WTO algorithm is executed for any withdrawal messages, so we study the number of withdrawal messages. We show in Figure 7 the number of withdrawal messages for the 10 routing tables. The maximum value is around 927496 for one day (for rrc00). This number of withdrawal means in average 927496/24/3600≈10.73 updates per second. Moreover this number is small compared with the number of total updates: 16956829 that means in average 196.26 update messages per second.

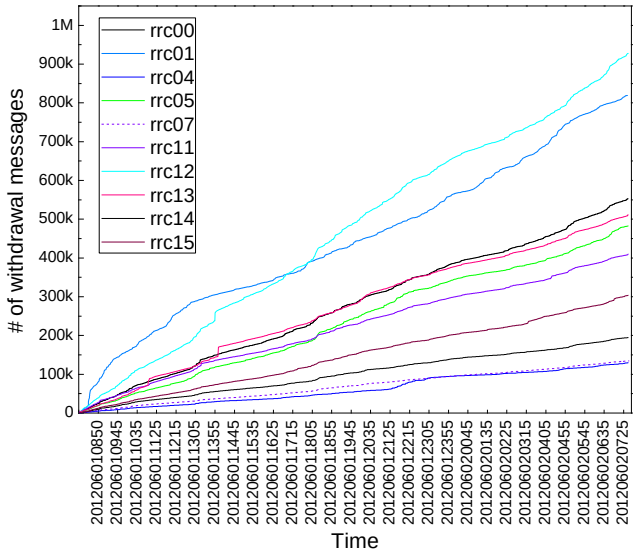


Figure 7. The number of updates in one day over 12 routers. The maximum value is around 927496 (rrc00) for one day. It is actually very tiny compared with the number of total updates: 16956829. The maximum 927496 withdrawal updates mean that 927496/24/3600≈10.73 withdrawal updates per second in average. For the most frequent updating router RRC00, there are 196.26 update messages per second in average.

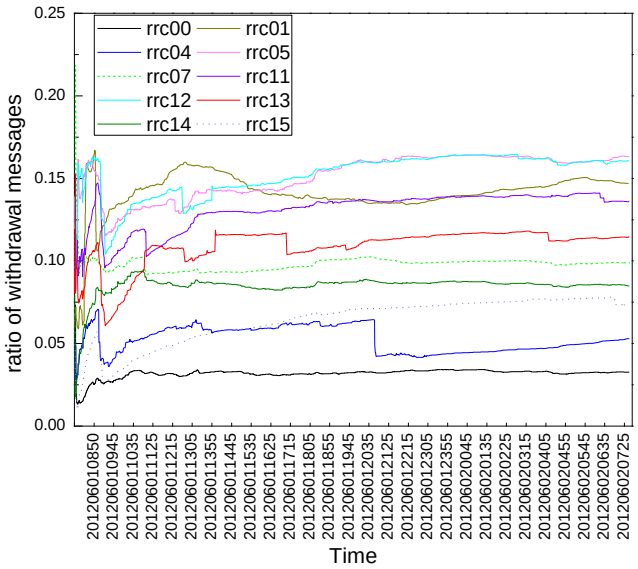


Figure 8. The ratio of the number of withdrawal update messages to that of total update messages. It can be observed that the ratio ranges from 0.03 to 0.16 with a mean of 0.1. This suggests that only 1/10 update messages are withdrawal. That means WTO is performed with the possibility of 1/10 in average.

As mentioned in Section I, the relative rarity of '*withdrawal messages*' makes WTO algorithm work well. To validate this, we plot the ratio of the number of withdrawal messages to the total update messages in Figure 8. It can be observed that the ratio ranges from 0.03 to 0.16 with a mean of 0.1. This

suggests that only 1/10 update messages are withdrawal. That means WTO is performed in average 1/10 of time.

As mentioned in Section III, WTO algorithm may cause domino effect. When the flag of updating node or the nearest ancestor node is true, WTO algorithm must traverse the sub-trie rooted at the updating node, then additional memory accesses are needed. Actually, because update messages are generally happening in '*Leaves*' [12], additional memory accesses are very low. We plot in Figure 9 the number of additional memory accesses,. Results show that the maximum number of additional memory accesses is 3620 over one day. Given a common DRAM working at 333MHz, 3620 additional memory accesses only need 3620/333000000=10.9us that is negligible.

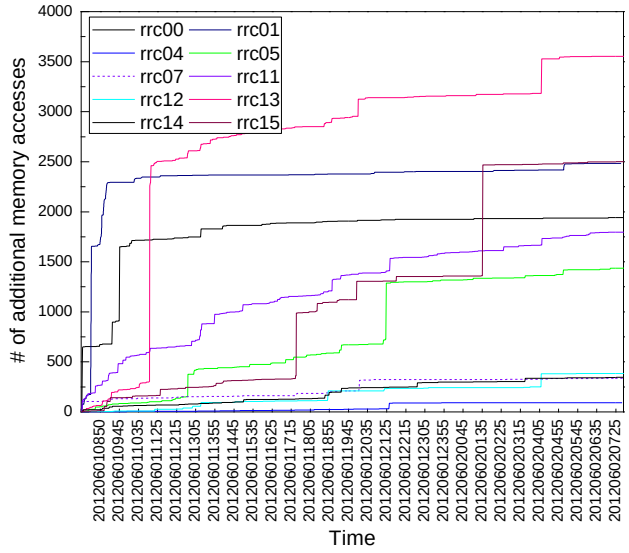


Figure 9. The number of additional memory accesses using WTO algorithm for 10 routing tables over one day. Because update messages have '*Leaves Characteristic*', the additional memory accesses are very few, and increases slower and slower. Results show that the maximum number of additional memory accesses is 3620 over one day.

To make a comparison, we plot the total memory accesses for updates including announcement and withdrawal messages over one day in Figure 10. The number of memory access number is about 0.4 billion, 110497 times of the additional 3620 memory accesses. That's to say, the negative effect of time overhead brought by WTO algorithm is only around 10^{-6} of the original update time.

We show in Figure 11, the number of additional memory accesses for each withdrawal after using WTO algorithm. It can be observed that in average only 0.001 additional memory accesses are needed.

The other negative effect of WTO algorithm is the introduction of *additional prefixes*. As WTO algorithm changes the withdrawal messages to announcement messages, there will be more prefixes after using WTO algorithm. The number of additional prefixes is an important metrics for WTO algorithm, as too many additional prefixes means larger routing table size, that might make WTO algorithm not practical.

According to Figure 7, there are 927496 update messages at most. One might think that the number of *additional prefixes* will be 927496. However, in practice the number is much smaller as only a small fraction of prefixes are frequently updated. We show in Figure 12 and 13, the number of additional prefixes observed after applying WTO.

The ratio of the number of additional prefixes to the total prefixes is shown in Figure 12. It shows that the ratio ranges from 0.002 to 0.04 with a mean of 0.01. This means that WTO algorithm results in average in only 0.01 additional prefixes produced for each withdrawal message.

As shown in Figure 13, the number of additional prefixes ranges from 979 to 13215 with a mean of 4050. This is much smaller than the number withdrawal messages (927496). If the router has enough memory, WTO algorithm can always work well. If the memory becomes insufficient, we can periodically perform a refresh, when the router is idle.

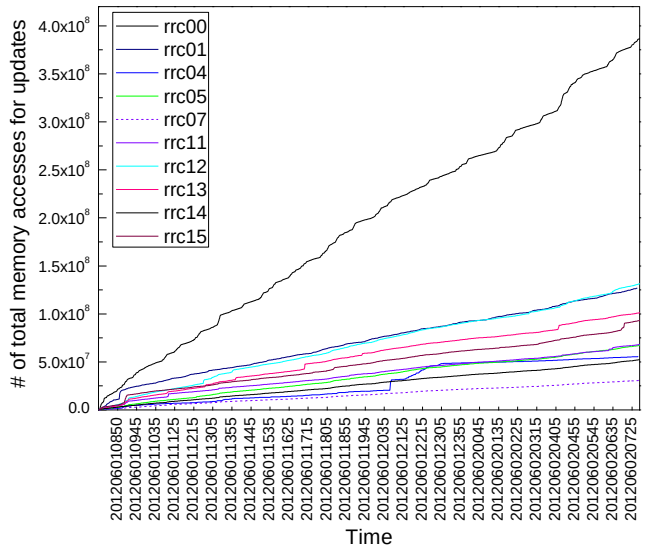


Figure 10. The number of memory accesses for update messages over one day. The maximum memory access number is about 0.4 billion, 110497 times of the additional 3620 memory accesses. That's to say, the negative effect of time overhead brought by WTO algorithm is only around 10^{-6} of the original update time.

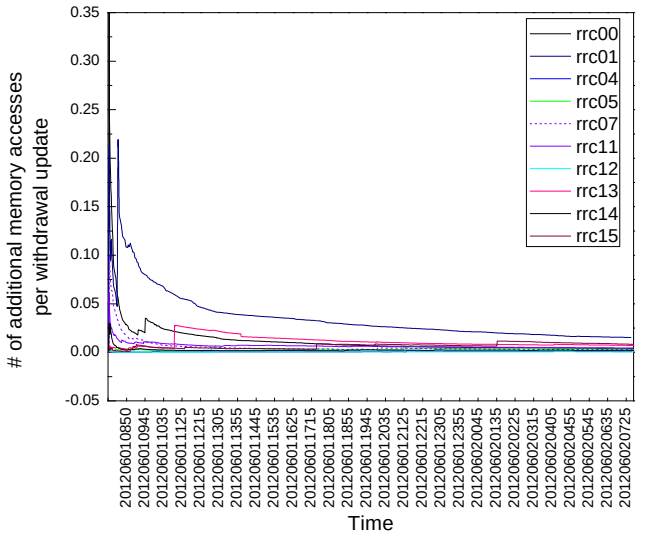


Figure 11. The number of additional memory accesses in average for each withdrawal message.

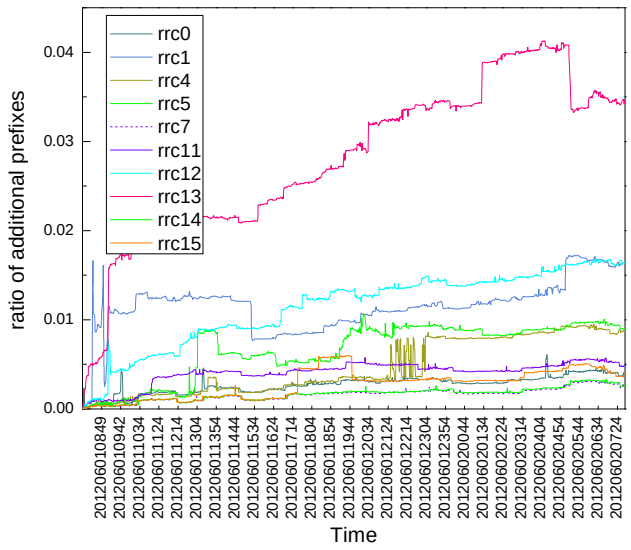


Figure 12. The ratio of the number of additional prefixes to that of total prefixes on 10 routing tables over one day. Results show that the ratio ranges from 0.002 to 0.04 with a mean of 0.01.

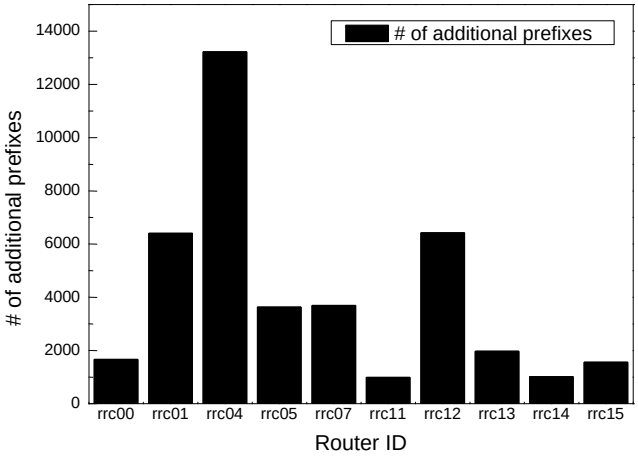


Figure 13. The number of additional prefixes on 10 routers over one day. The number of additional prefixes ranges from 979 to 13215 with a mean of 4050. This is much smaller than the number withdrawal messages (927496).

VI. CONCLUSION

In order to solve the issue raised by the usage of Bloom Filter based techniques for IP lookup, we propose WTO algorithm to solve the update problem for Longest Prefix Matching; In order to limit the worst case false alarm rate, we proposed DISBF algorithm to dynamically increase the size of Bloom Filters. We carried out experiments to evaluate the performance of WTO algorithm, and results show that they can overcome the shortcomings of Bloom filter-based solutions at the cost of negligible overhead. Note that DISBF algorithm can be applied to all situations using Bloom Filters, especially the frequently updating Bloom Filters. In our next work, we plan to apply DISBF algorithm to the Bloom Filter-based solutions for Exact Matching [8][9].

REFERENCES

[1] Ruiz-Sánchez M Á, Biersack E W, Dabbous W. Survey and taxonomy of IP address lookup algorithms. Network, IEEE, 2001, 15(2): 8-23.

[2] Sarang Dharmapurikar, Praveen Krishnamurthy David E. Taylor. Longest Prefix Matching Using Bloom Filters. In ACM SIGCOMM, 2003.

[3] Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422-426.

[4] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. False Negative Problem of Counting Bloom Filter. Knowledge and Data Engineering, IEEE Transactions on, 2010, 22(5): 651-664.

[5] Haoyu Song, Fang Hao, Murali Kodialam, T.V. Lakshman. IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards. In Proc. IEEE INFOCOM 2009: 2518-2526.

[6] Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended bloom filter: an aid to network processing. ACM SIGCOMM Computer Communication Review. ACM, 2005, 35(4): 181-192.

[7] Yan Qiao, Tao Li, Shigang Chen. One Memory Access Bloom Filters and Their Generalization. INFOCOM, 2011 Proceedings IEEE. IEEE, 2011: 1745-1753.

[8] Yu M, Fabrikant A, Rexford J. BUFFALO: bloom filter forwarding architecture for large organizations. Proceedings of the 5th international conference on Emerging networking experiments and technologies. ACM, 2009: 313-324.

[9] Dan Li, Henggang Cui, Yan Hu, Yong Xia, Xin Wang. Scalable data center multicast using multi-class bloom filter. Network Protocols (ICNP), 2011 19th IEEE International Conference on. IEEE, 2011: 266-275.

[10] FPGA Data Sheet.
http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[11] RIPE Network Coordination Centre.
http://www.ripe.net/data-tools/stats/ris/ris-raw-data.

[12] Tong Yang, Zhian Mi, Ruian Duan, Xiaoyu Guo, Jianyuan Lu, Shenjiang Zhang, Xianda Sun and Bin Liu. An Ultra-fast Universal Incremental Update Algorithm for Trie-based Routing Lookup. Network Protocols (ICNP), 2012 20th IEEE International Conference on. IEEE, 2012: 1-10.

[13] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. Internet Mathematics, vol. 1, no. 4, pp. 485–509, 2005.

[14] AS6447 BGP Routing Table Analysis Report.
http://bgp.potaroo.net/as6447/.