J Inf Process Syst, Vol.15, No.3, pp.550~569, June 2019
https://doi.org/10.3745/JIPS.04.0116

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Approaches for Improving Bloom Filter-Based Set Membership Query

HyunYong Lee* and Byung-Tak Lee*

Abstract

We propose approaches for improving Bloom filter in terms of false positive probability and membership query speed. To reduce the false positive probability, we propose special type of additional Bloom filters that are used to handle false positives caused by the original Bloom filter. Implementing the proposed approach for a routing table lookup, we show that our approach reduces the routing table lookup time by up to 28% compared to the original Bloom filter by handling most false positives within the fast memory. We also introduce an approach for improving the membership query speed. Taking the hash table-like approach while storing only values, the proposed approach shows much faster membership query speed than the original Bloom filter (e.g., 34 times faster with 10 subsets). Even compared to a hash table, our approach reduces the routing table lookup time by up to 58%.

Keywords

Additional Filters, Bloom Filter, False Positive Probability, Hash Table, Processing Time

1. Introduction

One of the basic functions in various IT areas is the membership query (i.e., determining a subset of a given element). For example, in computer networks, the membership query may mean finding an outgoing interface for a given packet [1,2], determining whether a given node is a malicious node [3], or finding a node who owns a requested file [4]. The membership query usually needs to be supported in a space-efficient manner because of the limit of storage space (e.g., a scarce on-chip fast memory for the performance issue [2] or overhead of traffic among network nodes [5]). Throughout this paper, we focus on the computer network as a representative example for clear presentation and understanding.

Bloom filter (BF) [6] has been one of the good choices for the space-efficient membership query. Supporting element insertion and element lookup based on a bit array, a BF supports the membership query for a set of elements (i.e., determining whether a given element belongs to the set or not). BF may falsely report that a given element belongs to the set even though the given element does not belong to the set. This error is called false positive. Even though the false positive probability of BF affects the performance of BF-based system (e.g., the false positive increases the number of accesses to the off-chip slow memory [1]), most existing BF-related work just tries to exploit BF in their target environments

Manuscript received November 29, 2017; first revision January 15, 2018; accepted March 15, 2018.

Corresponding Author: HyunYong Lee (hyunyonglee@etri.re.kr)

* Energy System Research Section, Honam Research Center, Electronics Telecommunications Research Institute (ETRI), Daejeon, Korea ({hyunyonglee, bytelee}@etri.re.kr)

while not improving BF.

In this paper, we try to improve BF in two aspects: false positive probability and processing time in the case where we need to find a subset among subsets for a given element. We first introduce two approaches for reducing the false positive probability. Unlike existing work that tries to reduce the false positive probability by manipulating BFs themselves, we introduce special type of additional BFs to handle false positives caused by original BFs. Keeping the correct subset information for false positives, the additional filters exclude candidate subsets falsely matched by the original BFs so as to find a correct subset. Through simulations, we first show that the use of the additional filters require certain amount of storage space (e.g., 227 kB is required in case of 1 M elements) to outperform the original BF. However, considering the amount of on-chip fast memory available nowadays (e.g., our experiment machine has 1.5 MB L2 data cache), we believe that the required storage space can easily be given. We also show that the proposed approaches can be applied to existing approaches (that try to reduce the false positive probability) to reduce their false positive probability further. This means that our approaches have the contribution to the existing approaches as well. Using Click modular router [7], we prove that the proposed approaches improve the routing table lookup performance (compared to the original BF) by handling most false positives within the on-chip fast memory.

Then, we introduce one approach for reducing the membership query time in a software system. In the software system where most things are done in a sequential manner, the membership query for a given element requires examination of BFs and thus the processing time increases as the number of filters increases. To achieve the fast membership query in a space-efficient manner, we take a hash table-based approach for the membership query (i.e., one hash computation for one membership query) while storing only values (i.e., subset IDs) without keys (i.e., elements). Through simulations, we show that our approach achieves the fast and constant processing time regardless of the subsets at the cost of slight additional storage space to achieve the same membership query error rate as the original BF. Using Click modular router, we prove that our approach achieves faster routing table lookup than a hash table (that is not space-efficient) by keeping whole routing table entries within the on-chip fast memory.

The rest of this paper is organized as follows. Section 2 describes background of this paper. Section 3 discusses approaches for reducing the false positive probability and Section 4 introduces one approach for improving the membership query speed. After Section 5 compares the proposed approaches, and Section 6 concludes this paper.

2. Background

2.1 Bloom Filter

A BF is a space-efficient probabilistic data structure that supports element insertion and membership query. A BF is an array of m bits that are set to 0 initially (Fig. 1(a)). To support the membership query for a set of n elements, the BF first needs to be generated by inserting n elements into the BF (Fig. 1(b)). For the element insertion, k hash functions are applied for each element to find k bits to be set to 1. The insertion of an element is completed when the corresponding k bits are set to 1. For the membership query (Fig. 1(c)), the same k hash functions are applied for a given element to find k bits to be examined. If all k bits are 1, the given element is regarded as a member of the set.

A BF may falsely report a non-member element as the member of the set. For example, if k bits for the non-member element (i.e., D of Fig. 1(c)) are set through the addition of elements of the set, that non-member element is falsely considered as the member of the set (i.e., *false positive*). The false positive probability can be expressed as

$$\left(1-e^{-k*\frac{n}{m}}\right)^{k}. \tag{1}$$

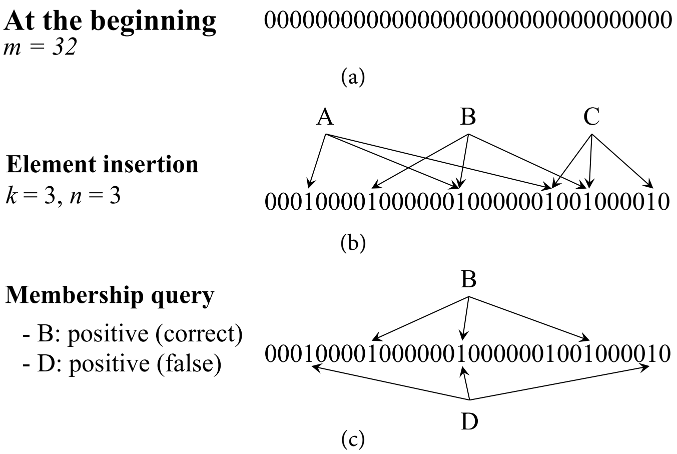

Fig. 1. Element insertion and membership query using a BF: (a) filter initialization, (b) element insertion, and (c) membership query.

2.2 Scope

Throughout this paper, we focus on the following environment. A given set S is divided into more than one subset and one element of S belongs to one subset. In this case, the membership query for a given element is to find a subset that the given element belongs to. We assume that additional data structures are used for the purpose of management (e.g., counting BF [5] that supports the element deletion can be managed in the large off-chip slow memory while BFs can be used for actual processing as shown in Fig. 2).

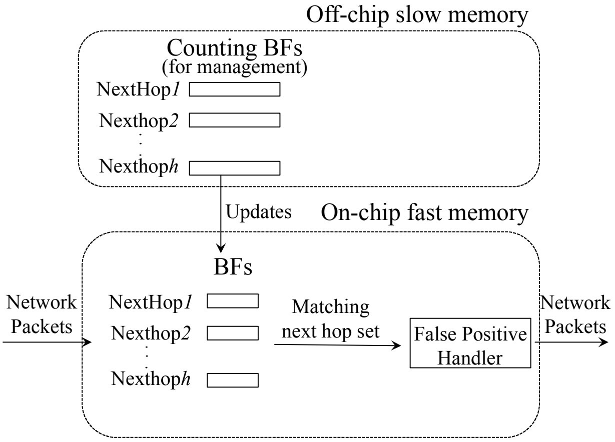

Fig. 2. BF-based routing table lookup for L2 switching.

In the above environment, we try to reduce the false positive probability and to improve the membership query speed. In that sense, our approach is not new. Here, we briefly compare our work with related existing works. In [8], the authors propose Cuckoo hashing-based approach, called Cuckoo filter to reduce the false positive probability with less storage overhead. Instead of using different architecture, we propose BF-based approach to achieve the same goal. In [9], the authors try to handle the issues with a large number of BFs. Even though [9] and we consider similar environment, we mostly aim at reducing the false positive probability. In [10], the authors propose the approach that requires only one base hash function to improve the membership query speed. In [11], the authors propose a BF variant that stores auxiliary information and supports faster membership query. While the authors [10,11] stick to BF-based approach, we try to achieve to the same goal by using hash table-based approach. In [12], the authors propose an approach for dynamic data sets. In particular, they try to improve the membership query speed while adjusting the filter size for dynamic data sets. On the other hand, we focus on the data sets that changes rarely.

2.3 An Example

To explain how BFs can be utilized, we take one example in the context of routing table lookup among various examples [13]. The routing table lookup is required to find an outgoing interface for a given packet in L3 routing [1] or L2 switching [2]. Because the memory access is one of the main performance bottlenecks, efficient use of the small on-chip fast memory is important in improving the routing table lookup performance.

BFs can improve the routing table lookup performance based on its space efficiency. In other words, BFs can support partial [1] or whole [2] routing table lookup process within the fast memory by representing a given routing table in a space-efficient manner. For example, in L2 switching (Fig. 2), a given routing table can be divided into subsets according to outgoing interface [2]. One BF for each subset (i.e., outgoing interface) is generated by inserting all the addresses that are forwarded to that interface (i.e., elements of a subset). Then, BFs are kept and used for the routing table lookup within the fast memory.

For a given packet, all BFs are examined. If the given packet is reported as a member of one BF, one matched BF means a correct outgoing interface. If the given packet is reported as a member of more than one BF, this means that some BFs cause false positives and the false positive handler handles the false positive. Supporting entire table lookup within the fast memory improves the table lookup performance compared to hash table.

3. Approaches for Reducing False Positive Probability

We first discuss two approaches using special type of additional BFs to reduce false positive probability. After introducing existing work, we describe our approaches.

3.1 Existing Work and Motivation

Few work tries to reduce the false positive probability of BF. One approach is to manipulate the number of hash functions for each element according to the request popularity distribution [14,15]. The rationale behind this approach is to allow a larger/smaller number of hash functions for more/less popular elements so as to decrease/increase the false positive probability. Even though the false positive probability of less popular elements increases, the actual number of false positives by those less popular elements does not change significantly because they are not queried frequently. On the other hand, popular elements that are queried frequently cause a smaller number of false positives. As a result, the total number of false positives decreases. Another approach is to manipulate specific bits [16]. Among bits that are related to false positives, bits that cause troublesome false positives are cleared (i.e., set to 0) so as to remove the possibility of the troublesome false positives.

Existing approaches have some limitations. First, monitoring the required information is needed, which causes space and computation overhead. Second, dynamic manipulation of BFs causes high computation overhead. Third, manipulating BFs for all elements is not an efficient way when considering the temporal locality (e.g., around one tenth of the entire IP prefixes accounts for more than 97% of traffic [17]). Fourth, clearing some bits causing troublesome false positives causes false negatives.

To reduce the false positive probability without causing above limitations, we introduce special type of additional BFs in addition to the original BFs. The additional BFs are used to keep the correct subset information of elements that cause false positive with the original BFs. The rationale behind this approach is to use the original BFs to find candidate subsets for a given element and to use the additional filters to exclude wrongly matched subsets from the candidate subsets. Please note that we do not use additional storage resources for the additional filters because BF is usually used in the case where storage resources are scarce. Detailed descriptions about our approaches will be described in Sections 3.2 and 3.3.

3.2 Using Additional Negative Bloom Filters

Our first approach is to introduce one negative BF (nBF) for one BF (Fig. 3). For the clear presentation, from now on, we use pBF for a normal BF and nBF for a corresponding negative BF in the proposed approach while we use BF for the original BF. We use term "negative" because the additional filter supports non-membership query. A nBF is generated by inserting elements that are not associated with a corresponding pBF, but cause false positives with the pBF. When a false positive happens (i.e., a given element matches with more than one pBFs), nBFs are used to exclude wrongly matched subsets.

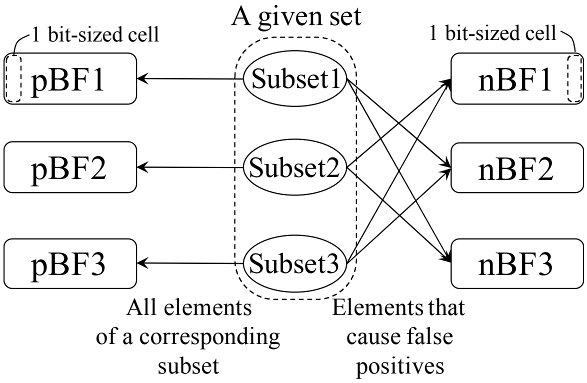

Fig. 3. A high-level illustration of the approach using nBFs.

3.2.1 Filter generation

Given subsets, pBFs are generated first. pBF_i is generated by inserting elements of $subset_i$. Then, nBFs are generated. nBF_i is generated by inserting elements that do not belong to $subset_i$, but cause false positives with pBF_i. In Fig. 3, cell indicates a basic unit of data to be manipulated.

3.2.2 Membership query

For a given element, pBFs are used for the membership query and nBFs are used for the non-membership query. In other words, the element x (∈ $subset_i$) should match with pBF_i and should not match with nBF_i to be considered as a member of $subset_i$. Please note that a BF including nBF never cause false negative. Therefore, match with nBF_i means that a given element is not a member of $subset_i$ even though it may falsely matches with pBF_i.

For membership query, pBFs are examined first. If only one pBF is matched, that pBF indicates a correct subset. If more than one pBF is matched, nBFs of the matched pBFs are examined to exclude wrongly matched subsets. If nBFs work correctly (i.e., no false positive), only one nBF of the matched pBFs should not match with the given element. However, because any BF including nBF can cause false positive, an element that belongs to subseti can match with nBF_i. In this case, a false positive caused by pBFs is not solved by nBFs.

3.2.3 Filter update

nBF_i needs to be updated when pBF_i changes. When a new element is added to $subset_i$ and pBF_i is updated, elements of other subsets that are not inserted into nBF_i need to be examined with the updated pBF_i to insert additional elements that cause false positive with the updated pBF_i into nBF_i. When an existing element is removed from $subset_i$ and pBF_i is updated, elements that are inserted into nBF_i need to be examined with the updated pBF_i to remove elements that do not cause false positive with the updated pBF_i anymore from nBF_i.

3.2.4 False positive probability

In the proposed approach, a false positive happens when pBFs cause a false positive and that false positive is not handled by nBFs of the matched pBFs. Considering that a BF never results in false negatives, a given element x (∈ $subset_i$) always matches with pBF_i and nBF_j ($j \neq i$) of wrongly matched pBF_j. However, the given element x may match with nBF_i. This means that the element x may be able to be considered as a non-member of $subset_i$. In this case, a false positive caused by pBFs cannot be handled by nBFs. As a result, the probability of that a false positive caused by pBFs is not handled by nBFs is equal to the false positive probability of nBF_i.

For a given element x, the probability of that the element x causes false positives with pBFs is

$$1-\prod_{l=1(l\neq i)}^{h}\left(1-fp(BF_l)\right)\approx\sum_{l=1(l\neq i)}^{h}fp(BF_l) \tag{2}$$

In Eq. (2), h is the number of subsets and $fp(BF_l)$ is

$$\left(1-e^{-k_l*\frac{n_l}{x*m_l}}\right)^{k_l} \tag{3}$$

In Eq. (3), k_l is the number of hash functions for *subset$_l$*, n_l is the number of elements of *subset$_l$*, and x^*m_l ($0 < x < 1$) is the number of bits allocated for *pBF$_l$*. Please note that *pBF$_l$* and *nBF$_l$* share the number of bits allocated for *BF$_l$* (i.e., m_l) and thus that the number of bits for *pBF$_l$* is smaller than that of *BF$_l$*. The false positive probability of *nBF$_i$*, *fp(nBF$_i$)* is

$$\left(1-e^{-k_i*\frac{n_i'}{(1-x)*m_i}}\right)^{k_i} \tag{4}$$

In Eq. (4), n_i' *is* *fp*(*pBF$_i$*) * (N – n_i). N is the total number of elements of a given set. Eq. (4) means that *fp*(*nBF$_i$*) is affected by the distribution of elements over subsets as well as *fp*(*pBF$_i$*). In achieving the minimum false positive probability of *fp*(*nBF$_i$*) with given settings (i.e., N, n_i, the total number of bits that are to be used by pBFs and nBFs, and the maximum number of hash functions), existing solvers such as IPOPT [18] can be used to find a proper value of x. Please note that x is around 0.95 in most our simulations.

Now, we briefly discuss the condition for our approach to outperform BFs only case (i.e., the original BFs without additional filters). For the purpose of comparison, we assume that the optimal number of hash functions (i.e., $k_i = m_i/n_i * ln2$) is used. Using the optimal number of hash functions, Eq. (1) can be transformed into $(1/2)^k$. Therefore, the condition for *fp(BF$_i$)* > *fp(nBF$_i$)* is

$$\left(\frac{1}{2}\right)^{\frac{m_i}{n_i}*ln2} > \left(\frac{1}{2}\right)^{\frac{(1-x)m_i}{n_i'}*ln2} \tag{5}$$

$$\frac{m_i}{n_i}*ln2 < \frac{(1-x)m_i}{n_i'}*ln2 \tag{6}$$

$$n_i' < (1-x)*n_i \tag{7}$$

$$fp(pBF_i)*(N-n_i) < (1-x)*n_i \tag{8}$$

$$fp(pBF_i) < \frac{(1-x)n_i}{N-n_i} \tag{9}$$

Eq. (9) means that the condition for our approach to outperform BFs only case is affected by m_i that influences *fp*(*pBF$_i$*) and the ratio of n_i to N. Therefore, given n_i and N, our approach needs certain amount of x*m_i for *pBFi* to satisfy Eq. (9).

To examine the condition for our approach to outperform BFs only case, we conduct simulations with settings described in Section 3.4. In this simulation, we first measure the required amount of storage space to outperform BFs only case by increasing the total amount of storage space by 10 kB. In Fig. 4, the numbers below scatter plot indicate the required amount of storage space in kB to outperform BFs only case for the given number of elements. This result shows that our approach requires some amount of storage space to outperform BFs only case. The amount of required storage space increases as the number of elements increases because *fp*(*BF*) is non-decreasing function of the number of elements. However, even in the case of 1 M elements, our approach just requires 227 kB storage space, which is relatively small compared to on-chip fast memory available today (e.g., our experiment machine has 1.5 MB L2 data cache).

During the same simulations above, we also measure the false positive probability of BFs only case at the time when our approach begins to outperform BFs only case (i.e., *fp$_{out}$*). In Fig. 4, the scatter plot shows that results. Interestingly, fp_{out} is over 9.0E-01 regardless of the number of elements. This result means that if BFs only case achieves the false positive probability lower than fp_{out}, then our approach can achieve lower false positive probability than that of BFs only case with the same amount of storage space.

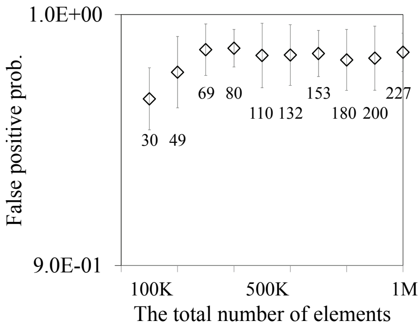

Fig. 4. The condition to outperform BFs only case.

3.3 Using an Additional Positive Bloom Filter

3.3.1 Motivation and approach

Even though the use of nBFs is likely to outperform BFs only case, the approach using nBFs has two limitations. One limitation is the update overhead. nBFs need to be updated whenever pBFs change. This means that nBFs may not be a good choice for a set that changes dynamically over time. Another limitation is inefficient use of filter. If an element causes false positives with other subsets, that element needs to be inserted into nBFs of those subsets. Considering that a larger number of elements lead to higher false positive probability, inserting one element into multiple nBFs is not good.

To overcome above limitations of using nBFs, we propose another approach (Fig. 5). While the new approach is similar to the use of nBFs in using the additional filter, the new approach is different from the use of nBFs in three ways. First, the new approach uses only one additional filter (called *sBF*). Second, sBF contains correct subset information to support membership query. This means that only one insertion into sBF is enough for handling a false positive. Third, sBF is populated dynamically to focus on active elements that are being queried. The detailed descriptions about the use of sBF are as follows.

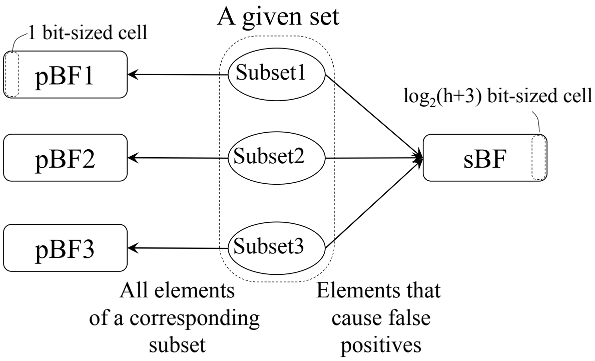
A given set
1 bit-sized cell
pBF1
pBF2
pBF3
Subset1
Subset2
Subset3
log$_2$(h+3) bit-sized cell
sBF
All elements of a corresponding subset
Elements that cause false positives

Fig. 5. A high-level illustration of the approach using sBF.

3.3.2 Data structure and filter generation

The role of additional filters including nBFs and sBF is to help find correct subset for a given element. A normal BF is not enough for this purpose because a normal BF just supports the membership query for a specific set. To return correct subset information with one filter, sBF stores an arbitrary value (i.e., correct subset information) instead of a binary value. For this, the size of a cell of sBF becomes $\lceil log_2(h+3)\rceil$, where h is the number of subsets and 3 is for NULL and CONFLICT state and a timing bit. Detailed description about this will be given later.

To focus on the active elements that are being queried, cells of sBF are set to NULL initially. If sBF does not contain required subset information for a given element that matches with more than one pBF, the insertion process begins. The insertion process is similar to that of a normal BF. To insert subset information (e.g., subset ID), k hash functions are applied to a given element to find k cells. Then, the values of k cells are modified. If the value of one cell is NULL or the same as the value to be inserted, the value of that cell is set to the value of correct subset information. In other cases, the value of a cell is set to CONFLICT, which means different values are allocated for the same cell and thus the value of this cell is not helpful. After sBF is properly populated, false positives caused by subsequent queries for the same element can be handled by sBF.

3.3.3 Membership query

For a given element, pBFs are first examined to find candidate subsets. If only one pBF is matched, that pBF indicates a correct subset. If more than one pBF is matched, then, sBF is examined to get correct subset information. If sBF is properly populated and works correctly (i.e., no false positive), sBF returns the correct subset information.

For the examination of sBF, k hash functions are applied to a given element to find k cells to be examined. To return a value, the values of k determined cells are examined. If the values of all cells are CONFLICT, CONFLICT is returned. If the values of cells (excluding cells whose value is CONFLICT) are different, the return value is NULL. This means that a given element is not inserted into sBF yet and thus the insertion process (described in Section 3.3.2) is done. In above two cases, the false positive caused by pBFs cannot be handled. If the values of cells (excluding cells whose value is CONFLICT) are same, that value is returned. If the returned value is not one of the matched pBFs, this means that a given element is not inserted into sBF yet and thus the insertion process is done. If the returned value is one of the candidate subsets identified by pBFs, the returned value indicates the correct subset.

3.3.4 Filter update

One of the main goals of using sBF is to focus on the active elements that are being queried. Therefore, we need to delete inactive elements that are not queried for a long time. For this purpose, we allocate one bit-sized timing bit for each cell of sBF. The timing bit is set to 0 initially or whenever a global timer (e.g., Linux kernel timer) goes off. When cells are accessed and one value is returned, the timing bits of the accessed cells are set to 1. Whenever the global timer goes off, values of cells whose timing bit's value is 0 are set to NULL. This timing-based approach can update sBF gracefully to keep correct subset information for active elements while not causing noticeable management overhead.

3.3.5 False positive probability

A false positive happens when a false positive caused by pBFs is not handled by sBF. Assuming that sBF is properly populated for active elements that are being queried, a false positive happens when sBF returns CONFLICT. The false positive probability of the proposed approach is

$$(\sum_{i=1}^{h} fp(BF_i)) * fp(sBF) \tag{10}$$

In Eq. (10), *fp*(*sBF*) is the probability of that sBF returns CONFLICT. *fp*(*sBF*) is

$$\left(1 - e^{-k*\frac{n_s}{m_s}}\right)^k \tag{11}$$

In Eq. (11), m_s is $m/\lceil log_2(h+3)\rceil$, where m is the number of bits for sBF. n_s is the number of elements to be inserted into sBF, i.e., $(\sum_{i=1}^{h} fp(BF_i)) * N_{ACTIVE}$, where N_{ACTIVE} is the number of current active elements. Eq. (11) means that the false positive probability of the proposed approach is affected by $\mathrm{N_{ACTIVE}}$ with given m. To examine the effect of N_{ACTIVE} on the false positive probability, we conduct simulations with the settings described in Section 3.4. From the simulations, we found that the false positive rate of sBF increases as the percentage of N_{ACTIVE} increases. The false positive rate increases from 3.33E-08 to 1.87E-05 as the percentage of N_{ACTIVE} increases from 20% to 40%.

To minimize the false positive probability of the proposed approach, we need to find a proper value for m with given the total number of elements, the number of active elements, the maximum number of hash functions, the number of subsets, and the total amount of storage space. Like the case of the use of nBFs, existing solvers such as IPOPT can be used for solving this problem. Please note that m is around 10% of the given total amount of storage space in our simulations.

3.4 Evaluation

For simulations, we use different values for the total number of elements (i.e., from 100 K to 2 M), the number of subsets (i.e., from 10 to 100), the total amount of storage space (i.e., from 100 kB to 1 MB), and the ratio of $\mathrm{N_{ACTIVE}}$ to the total number of elements (i.e., from 5% to 40%). The maximum number of hash functions is 8. Elements are evenly distributed over subsets. For the purpose of analysis, nBFs and sBF are pre-populated. The total number of queries is equal to the number of given elements. The number of queries for subsets is same. The results are the average across 50 runs. The false positive rate is calculated as the ratio of the number of false positives to the total number of queries. In the following figures and manuscripts, BF, nBF, and sBF indicate BFs only case and the proposed approaches with the additional filters, respectively.

3.4.1 Performance study

Fig. 6 shows the change of false positive rate with regard to the storage space. In Fig. 6, "w/o" and "w/" indicate the case of the proposed approaches without the additional filters (i.e., only pBFs smaller than the original BFs) and with additional filters, respectively. In the proposed approaches, pBFs and the additional filters share the given storage space. As a result, the proposed approaches without the additional filters cause a larger number of false positives than BF. On the other hand, the proposed approaches with the additional filters begin to cause a smaller number of false positives than BF when enough storage space is given as mentioned before. In this simulation setting, the proposed approaches begin to outperform BF from 100 kB storage space.

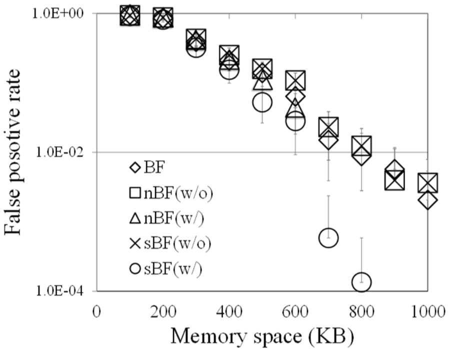

Fig. 6. False positive rate w.r.t. the storage space (600 K elements, 10 subsets, and 10% N_{ACTIVE}).

The false positive rate of the proposed approaches decreases faster than that of BF as the amount of storage space increases (Fig. 6). This happens because increasing storage space decreases (i) the number of elements to be managed by the additional filters by decreasing the overall false positive probability of pBFs and (ii) the false positive probability of the additional filters by increasing storage space for the additional filters. As a result, nBF does not show false positives from 700 kB and sBF does not show false positives from 900 kB.

Fig. 7 shows the change of false positive rate with regard to the number of subsets. The false positive rate of BF is heavily affected by the number of subsets. This is due to the fact that the overall false positive probability of BF is the sum of false positive probability of each BF. Therefore, the false positive rate of BF increases linearly as the number of subsets increases. On the other hand, the additional filters of the proposed approaches can handle the increasing false positives properly and thus the false positive rate of nBF and sBF increases much more slowly than that of BF. The false positive rate of BF/nBF/sBF increases from 3.65E-04 to 2.30E-02/from 0 to 3.24E-04/from 0 to 1.02E-03 as the number of subsets increases from 5 to 100. The false positive rate of sBF is more subject to the number of subsets than that of nBF because the number of cells of sBF is affected by the number of subsets as we mentioned before. As a result, the false positive rate of sBF increases slightly faster than that of nBF.

We compare three approaches in another aspect, the number of bits per element required to achieve the target false positive rate (Fig. 8). In this simulation, we increase the storage space by 10 kB until each approach achieves the target false positive rate. The required number of bits per elements increases somewhat linearly in all three approaches as the target false positive rate decreases until certain point. From certain target false positive rate, the number of bits per element does not increase because each approach can achieve lower target false positive rate with the same number of bits per element. nBF and sBF do not require additional bits per elements from 1.0E-03 and 1.0E-04 while BF does not require additional bits per element from 1.0E-07. This means that nBF and sBF achieve the target false positive rate with a smaller number of bits per element than BF.

We compare our approaches with existing approaches [14,15]. In Fig. 9, we only show the results of nBF case because sBF shows the same result patterns. In this simulation, nBF outperforms wBF [14] and dBF [15] outperforms nBF. However, their results may change with different settings. For example, the false positive probability of existing approaches is mainly affected by the query popularity distribution.

On the other hand, the false positive probability of our approaches is affected by the number of active addresses and other things. Therefore, it is hard to say which approach is the best. Instead, what we want to say is that our approaches can be used with existing approaches to reduce their false false positive probability further. For example, wBF with nBFs show no false positives from 500 kB and dBF with nBFs show no false positives from 200 kB. This means that our approaches have the contribution to the existing approaches as well as the original BF.

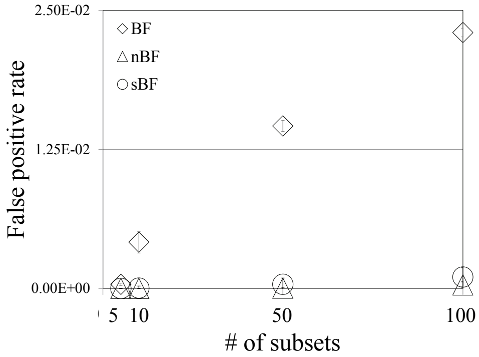

Fig. 7. False positive rate w.r.t. the number of subsets (1 MB storage space, 600 K elements, and 10% N_{ACTIVE}).

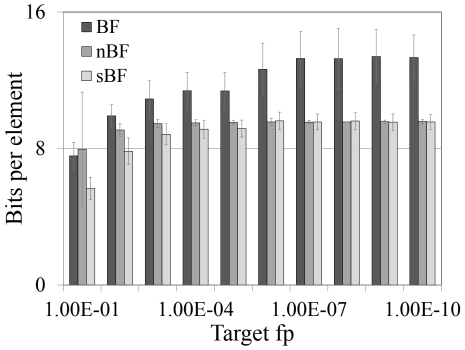

Fig. 8. Bits per element for achieving the target false positive rate (1 MB storage space, 600 K elements, 10 subsets, and 10% N_{ACTIVE}).

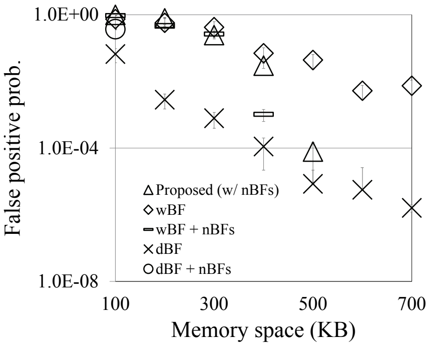

Fig. 9. Comparison with existing approaches (nBF).

3.4.2 Routing table lookup

To examine how much our approaches improve performance of BF-based applications, we implement BF-based routing table lookup using Click modular router [7]. We modified EtherSwitch element to implement the routing table lookup with BF and our approaches. For the purpose of comparison, a routing table implemented using a hash table in EtherSwitch is accessed when a false positive happens in BFs only case and a false positive is not solved by nBF or sBF in the proposed approaches. The number of elements (i.e., L2 MAC addresses as routing table entries) is 1 M and the number of outgoing interfaces is 10. N_{ACTIVE} is 100 K and the number of queries is 10 M. The destination MAC address is randomly chosen from N_{ACTIVE} addresses. The storage space used for filters is 640 kB so that filters can be fit into the 1.5 MB-sized L2 cache of our experiment machine.

Table 1 shows the average routing table lookup time in ns. In case of successful lookups (i.e., no false positives), three approaches show almost similar results. However, in case of false positives, three approaches show different results. In this case, BFs only case shows poor routing table lookup performance because it needs to access the hash table for 1 M MAC addresses that is too big to be kept in the on-chip fast memory. Therefore, a false positive in BFs only case causes accesses to the off-chip slow memory. On the other hand, nBF and sBF show better routing table lookup performance. Even though the use of additional filters increase the number of false positives by pBFs compared to BFs only case, nBF and sBF handle most false positives within the on-chip fast memory. For example, the use of sBF increases the number of false positives by pBFs by 15%. However, sBF handles 95% of false positives within the fast memory. When a false positive is handled by the additional filters, the routing table lookup time is faster than BFs only case because that process is done within the fast memory. As a result, the average routing table lookup performance in case of false positives is better than BFs only case. Because of this reason, nBF and sBF reduce the routing table lookup time by 20% and 28% compared to BFs only case.

Table 1. Routing table lookup time (unit: ns)

	BFs only	nBF	sBF
Successful lookups	83	84	85
False positives	210	Solved: 112 Not solved: 295 Total average: 114	109 294 111
Total average	120	95	86

The routing table lookup performance of sBF is slightly better than that of nBF. This is due to the fact that sBF just needs to examine one additional filter while nBF may need to examine more than one additional filter. The routing table lookup performance when a false positive is not handled by nBF and sBF is worse than the case of BFs only case. This is due to the fact that L2 cache hit rate is affected by the number of accesses. In other words, BFs only case shows better performance in this case because it accesses the off-chip slow memory more than nBF or sBF.

3.4.3 Summary

The proposed approaches that exploit the additional filters have three attractive points. First, the proposed approaches reduce the false positive probability compared to the original BF and thus improves the performance of BF-based applications. Second, the use of additional filters does not cause the identified limitations of existing approaches described in Section 3.1. Third, the additional filters can be used for existing approaches to reduce the false positive probability of those approaches further.

nBF outperforms sBF in most cases, particularly when the percentage of N_{ACTIVE} is high. Therefore, nBF is more suitable for the case where subsets change rarely. On the other hand, in the case where subsets change frequently and the percentage of N_{ACTIVE} is low (like Internet environment where around 10% of IP prefixes accounts most traffic [17]), sBF is more suitable. However, if the update overhead of nBF can be properly managed while not causing performance degradation, nBF would also be a good choice for such a case.

4. An Approach for Improving Processing Speed

4.1 Motivation

Most BF-related approaches have two requirements: space efficiency and membership query. However, one problem (i.e., processing overhead) may happen when BF-based system is implemented using software. In the software-based system, required processing (i.e., k hash operations for each filter) is done sequentially. As a result, the processing time increases somewhat linearly as the number of subsets and the number of hash functions used increase. This processing overhead may become problematic for time-sensitive jobs (e.g., routing table lookup and traffic monitoring). In other words, in those time-sensitive jobs, there are three requirements: space efficiency, membership query, and fast processing time. To this end, we propose one approach to satisfy those three requirements for the time-sensitive jobs.

4.2 Hash Table-based Approach

One simple approach to support fast membership query is to use a hash table. In the hash table, one hash computation is enough to find a bucket (i.e., basic unit of a hash table for storing data) to be examined. However, the hash table is not space-efficient. Most hash table implementations store both key (e.g., element itself) and value (e.g., subset ID) to handle hash collisions (i.e., the case where more than one element is mapped to the same bucket). As a result, the space overhead of the hash table increases as the number of elements and the size of element in bits increase.

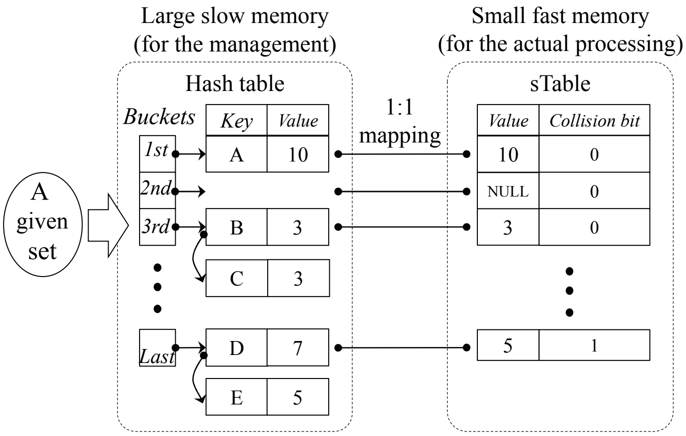

Fig. 10. A high-level illustration of sTable.

In this paper, we try to achieve the space efficiency while supporting fast membership query. For this, our proposed approach (called sTable) keeps value only without keys (Fig. 10). As a result, the space overhead of sTable is not affected by the number of elements and the size of element in bits, but affected by the number of subsets that affects the required number of bits to represent subsets. The detailed descriptions about sTable are as follows.

4.2.1 Data structure and generation

sTable consists of buckets like a hash table. Each bucket includes a value (e.g., subset ID of an element mapped to this bucket or NULL as the initial value) and 1 bit-sized collision bit. Therefore, the size of one bucket is $\lceil \log_2(h+2) \rceil$ bits, where h is the number of subsets. Collision bit is used to indicate the status of a corresponding bucket in terms of hash collisions (i.e., 0 means no hash collision and 1 means hash collision). Because sTable does not store key that is required to handle hash collisions, sTable cannot handle hash collisions. In other words, sTable achieves the space efficiency at the cost of some membership query errors (i.e., sTable cannot return correct subset information for a given element when the given element is mapped to a bucket where hash collisions happen). Instead, sTable utilizes the collision bit to indicate (i) the value of a bucket may not be correct and/or (ii) additional operation is required to find correct subset information. For example, the additional operation for handling membership query error of sTable may be examining the additional data structures managed separately (e.g. a hash table kept in large off-chip slow memory). Given the storage space M (in bits) and the number of subsets h, the number of buckets of sTable is $M/\lceil \log_2(h+2) \rceil$.

The content of sTable's buckets is determined by a corresponding hash table that is kept in large off-chip slow memory for the purpose of the management (Fig. 10). The hash table is first generated based on a given set. In this case, the key is element itself and the value is a corresponding subset ID. For the purpose of easy management, the number of buckets of the hash table is set to the number of buckets of sTable. In other words, buckets of sTable and the hash table have 1:1 mapping. The content of i^{th} bucket of sTable is determined by the content of i^{th} bucket of the hash table. If the bucket of the hash table does not contain any elements (i.e., 2nd bucket in Fig. 10), the value and the collision bit of the bucket of sTable are set to NULL and 0, respectively. If the bucket of the hash table contains one element (i.e., 1st bucket) or if the values of elements in the bucket of the hash table are same (i.e., 3rd bucket), the value and the collision bit of the bucket of sTable are set to the value of the hash table's bucket and 0, respectively. In other cases (i.e., the case where hash collisions happen, the last bucket), the value of the bucket of sTable is randomly set to one of the values of the hash table's bucket while the collision bit is set to 1.

4.2.2 Membership query

The membership query with sTable is done as in a hash table. One hash function is applied to a given element to find a bucket to be examined. If the collision bit of the determined bucket is 0 and the value of the bucket is not NULL, the value of the bucket indicates correct subset information. On the other hand, if the collision bit is 1, the value of the bucket may or may not indicate correct subset information. Therefore, if some membership query error is allowable, the value of the bucket may be able to be used. If additional operations are available to handle this membership query error at the cost of additional overhead (e.g., access to slow memory to examine the hash table) and the additional overhead is allowable, additional operations can be done to find correct subset information.

4.2.3 Update

When subsets change, the hash table is first updated. If contents of hash table's buckets are changed (i.e., a new element is added to an empty bucket, different values of the same bucket become same, or same values of the same bucket becomes different), contents of corresponding buckets of sTable are updated.

4.2.4 Membership query error probability

In sTable, membership query error happens when a given element is mapped to a bucket whose collision bit is 1. The membership query error probability can be calculated using birthday problem [19] as follows. Given b buckets for sTable and n elements, the membership query error probability can be approximately expressed as

$$n/b \tag{12}$$

Eq. (12) means that given the number of elements, the membership query error probability is mainly affected by the number of subsets and the storage space that affect b.

4.3 Evaluation

4.3.1 Comparison with BF

Using the simulation settings described in Section 3.4, we conduct simulations. We first measure the change of the membership query error rate with regard to the number of subsets (Fig. 11). As we described before, the error rate of BF increases somewhat linearly as the number of subsets increases. On the other hand, the error rate of sTable increases logarithmically as the number of subsets increases. The number of subsets affects the error rate of sTable by influencing the number of buckets, but the number of buckets of sTable does not decrease linearly as the number of subsets increases. Please note that the size of one bucket of sTable is $\lceil \log_2(h+2) \rceil$. As a result, the error rate of sTable increases more slowly than that of BF.

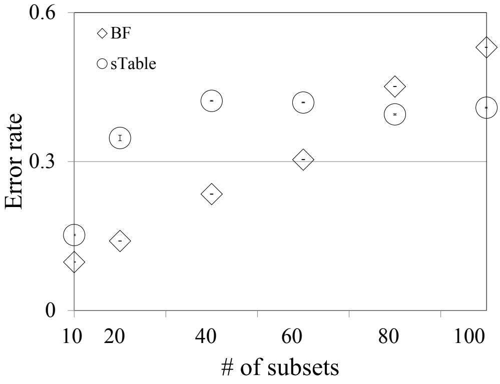

Fig. 11. The membership query error rate w.r.t the number of subsets (1 MB storage space, 600 K elements, and 10% N_{ACTIVE}).

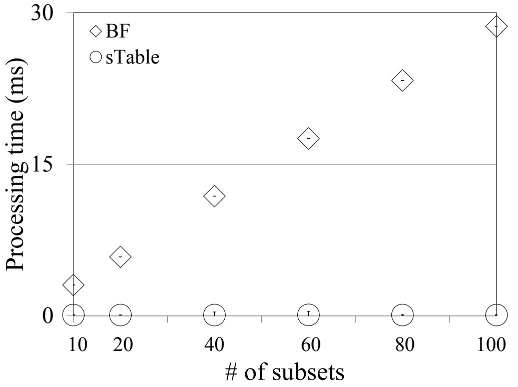

Fig. 12. The processing time w.r.t the number of subsets (1 MB storage space, 600 K elements, and 10% N_{ACTIVE}).

We measure the effect of the number of subsets on the processing time (Fig. 12). The processing time of BF increases linearly as the number of subsets increases because the number of filters to be examined increases (i.e., from 2.6 ms for 10 subsets to 26 ms for 100 subsets). On the other hand, sTable shows somewhat constant processing time regardless of the number of subsets and the processing time is much smaller than that of BF. sTable shows 0.076 ms and 0.081 ms processing time for 10 subsets and 100 subsets, respectively.

4.3.2 Comparison with hash table

To examine how much sTable improves performance compared to a hash table, we conduct the experiment using the same settings described in Section 3.4 except the following. We use 30 outgoing interfaces and 1024 kB is used for sTable. Table 2 shows the results. The successful routing table lookup with sTable (i.e., 85% of packets are handled within the fast memory) is much faster than that with the hash table. This is due to the fact that sTable handles packets within the fast memory while routing table lookup with the hash table causes accesses to the slow memory. As a result, sTable reduces the routing table lookup time by 58% compared to the hash table.

Table 2. Comparison of sTable and hash table

	Average time (ns)
Hash table	210
sTable	
Solved	37
Not solved	229
Total average	67

We also measure the required amount of storage space to achieve the target error rate (Fig. 13). For the hash table, we assume that the size of key is 48 bits (e.g., MAC address). Please note that the hash table always returns a correct result and thus requires the same amount of storage space regardless of the target error rate. On the other hand, BF and sTable require a larger amount of storage space to achieve the lower target error rate. The required amount of storage space of BF and sTable increases linearly and logarithmically as the target error rate decreases, respectively. As a result, the difference between the required amount of storage space for BF and sTable decreases as the target error rate decreases.

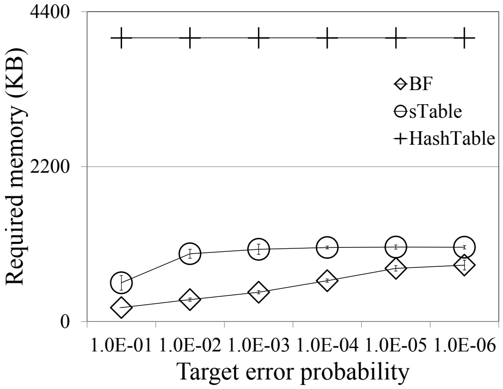

Fig. 13. The required amount of storage space for achieving the target error probability (1 MB storage space, 600 K elements, 10 subsets, and 10% N_{ACTIVE}).

Above results first show that sTable achieves the fast and somewhat constant processing time at the cost of additional storage space. However, the required amount of storage space is much smaller than that of the hash table. In addition, unlike the hash table where the space overhead is affected by the size of key and the number of subsets, the space overhead of sTable is only affected by the number of subsets. The amount of additional storage space decreases as the target error rate decreases. Therefore, sTable may be more preferable than BF when the target error rate is low and the processing time is important.

5. Discussion

We briefly compare the proposed three approaches as follows in the hope that the comparison is helpful to determine which one is proper for a given environment. In the following, $A < B$ means A *outperforms* B. For the sake of simplicity, we assume that enough storage space is given for our approaches so that our approaches outperform BFs only case.

- Space overhead: nBF < sBF < BF < sTable
- Error rate: nBF < sBF < BF < sTable
- Processing overhead: sTable < sBF < nBF < BF
- Management overhead: sTable < BF < sBF < nBF

Space overhead: For the same target membership query error rate, nBF and sBF require less amount of storage space than BF. sTable achieves the same error rate at the cost of slight additional storage.

Error rate: Given the same amount of storage space, nBF and sBF cause a smaller number of errors than BF, which causes a smaller number of errors than sTable.

Processing overhead: sTable provides the fastest membership query. On the other hand, the relationship between BF and nBF/sBF depends on the case. If there is no false positives, BF, nBF, and sBF may show the same membership query speed because the examination of additional filters is not required. If there are false positives and additional operations (e.g., accessing the off-chip slow memory) are required, nBF and sBF may show better membership query speed than BF. In this case, nBF may take slightly more time than sBF because it may need to examine more than one filter.

Management overhead: To insert or delete elements, sTable just needs one memory access for the bucket while BF-based approaches need to access *k* bits. The additional filters cause additional management overhead. In particular, nBF requires higher update overhead than sBF because it uses more than one additional filter.

6. Conclusion

Space-efficient membership query is useful in many areas. BF has been one of the good choices for the space-efficient membership query. Therefore, improving BFr may have noticeable impact on applications using BF. In this paper, we propose two approaches that utilize additional filters to reduce false positive probability. Simulations and experiments show that using additional filters reduce the false positive probability of existing approaches as well as the original BF and improve the BF-based routing table lookup performance. We also introduce one approach for improving membership query speed. Keeping values only, our approach archives fast and space-efficient membership query. Evaluation results prove that our approach is feasible in improving the membership query speed in a space-efficient manner.

Acknowledgement

This work was supported by ETRI R&D Program (No. 19ZK1140), funded by the government of Korea.

References

[1] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, “Longest prefix matching using bloom filters,” *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 2, pp. 397-409, 2006.

[2] M. Yu, A. Fabrikant, and J. Rexford, “BUFFALO: bloom filter forwarding architecture for large organizations,” in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, Rome, Italy, 2009, pp. 313-324.

[3] H. Wu, D. Yang, and G. Zhang, “SybilBF: defending against Sybil attacks via Bloom filters,” *ETRI Journal*, vol. 33, no. 5, pp. 826-829, 2011.

[4] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, “Informed content delivery across adaptive overlay networks,” *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 47-60, 2002.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281-293, 2000.

[6] 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.

[7] Click modular router, https://github.com/kohler/click.

[8] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo filter: Practically better than bloom,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, Sydney, Australia, 2014, pp. 75-88.

[9] A. Crainiceanu, “Bloofi: a hierarchical Bloom filter index with applications to distributed data provenance,” in *Proceedings of the 2nd International Workshop on Cloud Intelligence*, Trento, Italy, 2013.

[10] J. Lu, T. Yang, Y. Wang, H. Dai, L. Jin, H. Song, and B. Liu, “One-hashing bloom filter,” in *Proceedings of 2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, Portland, OR, 2015, pp. 289-298.

[11] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li, “A shifting bloom filter framework for set queries,” *Proceedings of the VLDB Endowment*, vol. 9, no. 5, pp. 408-419, 2016.

[12] Y. Liu, X. Ge, D. H. C. Du, and X. Huang, “Par-BF: a parallel partitioned Bloom filter for dynamic data sets,” *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 259-275, 2016.

[13] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, “Theory and practice of bloom filters for distributed systems,” *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131-155, 2011.

[14] M. Zhong, P. Lu, K. Shen, and J. Seiferas, “Optimizing data popularity conscious bloom filters,” in *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, Toronto, Canada, 2008, pp. 355-364.

[15] J. Bruck, J. Gao, and A. Jiang, “Weighted bloom filter,” in *Proceedings of 2006 IEEE International Symposium on Information Theory*, Seattle, WA, 2006, pp. 2304-2308.

[16] B. Donnet, B. Baynat, and T. Friedman, “Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives,” in *Proceedings of the 2006 ACM CoNEXT Conference*, Lisbon, Portugal, 2006.

[17] C. Kim, M. Caesar, A. Gerber, and J. Rexford, “Revisiting route caching: the world should be flat,” in *Passive and Active Network Measurement*. Heidelberg: Springer, 2009, pp. 3-12.

[18] COIN-OR Interior Point Optimizer (IPOPT) [Online]. Available: https://github.com/coin-or/Ipopt.

[19] D. Wagner, “A generalized birthday problem,” in *Advances in Cryptology-CRYPTO* 2002. Heidelberg: Springer, 2002, pp. 288-304.

HyunYong Lee https://orcid.org/0000-0002-0615-4241

He received the M.S. and Ph.D. degrees in computer science from the Gwangju Institute of Science and Technology (GIST), Korea, in 2003 and 2010, respectively. He is a currently senior researcher of Electronics Telecommunications and Research Institute (ETRI), Korea. His research interests include deep learning for renewable energy system.

Byung-Tak Lee https://orcid.org/0000-0003-1372-4561

He received his B.S. degree in electronic engineering from Yonsei University, Seoul, Korea, in 1992 and his M.S. and Ph.D. degrees in electrical and electronic engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1994 and 2000, respectively. From 2000 to 2004, he was a principal R&D engineer at LG Electronics, Seoul, Korea, where he was engaged in the area of 1.6 Tbps long-haul dense wavelength division multiplexing systems. Since 2005, he has been a senior research engineer at ETRI, Gwangju, Korea. His current research interests include Internet of Things and data analytics.