Reviriego, P., et al. Adaptive one memory access bloom filters. In: *IEEE Transactions on Network and Service Management, 19(2), June 2022, Pp. 848-859*

DOI: https://doi.org/10.1109/TNSM.2022.3145436

Adaptive One Memory Access Bloom Filters

Pedro Reviriego[1], Alfonso Sánchez-Macian[2], Ori Rottenstreich[3] and David Larrabeiti[1]

***Abstract*—Bloom filters are widely used to perform fast approximate membership checking in networking applications. The main limitation of Bloom filters is that they suffer from false positives that can only be reduced by using more memory. Recently, exploiting the fact that in many cases a few elements are queried many times on the filter, has been proposed to increase accuracy in other filters with a similar functionality. The idea is that once an element returns a false positive, the filter is adapted so that future queries for that element will not return a false positive. However, to the best of our knowledge, so far no scheme has been presented to support dynamic adaptation in Bloom filters. In this paper, one memory access Bloom filters are used to design an adaptation scheme that can effectively remove false positives while completing all queries in a single memory access. The proposed filters are well suited for scenarios on which the number of memory bits per element is low and thus complement the existing adaptive cuckoo filters that are not efficient in that case. The evaluation results using packet traces show that the proposed adaptive Bloom filters can significantly reduce the false positive rate in networking applications with the single memory access. In particular, when using as few as four bits per element, false positive rates below 5% are achieved.**

***Index Terms*—Computer networks; Data Structures; Bloom Filter.**

I. Introduction

In many networking applications, approximate membership checking is used to quickly determine if further processing is needed for a data element or if it can be safely discarded. This initial checking can significantly improve performance when most of the elements do not need full processing. For example, before accessing an element stored in a slow external memory, a fast false-negative-free check using a faster memory can speed up the system and reduce bandwidth requirements for the external memory. This fast checking has been traditionally implemented with Bloom filters [1] that are commonly used in networking systems [2], [3]. More recently, the cuckoo filter [4], [5] has also been widely adopted in a similar manner, offering some advantages over Bloom filters like the support of deletions or a lower false positive probability when the number of memory bits per element stored in the filter is large [6]. Cuckoo filters have, however, also disadvantages, for example in terms of a relatively high false positive rate when occupancy is low and most cells are empty [6] and in lookup speed [7]. In fact, Bloom filters can be implemented in such a way that lookups only require one memory access [8], whereas cuckoo filters require two memory accesses for negative lookups that would be the majority in practical scenarios. Additionally, cuckoo filters can suffer from insertion failures at low occupancy when the number of memory bits per element is below six, which limits their use for memory constrained applications [4]. For Bloom filters, constructions that completely avoid false positives have been recently suggested but they imply hard restrictions on the size of the represented set and the size of the universe from which elements are taken [9], [10].

An interesting observation is that, in many cases, the same elements are repeatedly queried in the filter. For example, when monitoring a small fraction of the flows on a link, a filter can be used to quickly determine if a packet belongs to one of the flows being monitored. In this case, elements correspond to flows and thus each element (flow) is queried for each packet in the flow and flow distribution is biased [11]. When the frequency or popularity of the elements is known in advance, the number of hash functions used for insertions and queries can be set according to the likelihood of the element to reduce the false positive rate [12], [13]. However, this approach requires additional offline information on the elements that is not commonly available or easy to obtain in networking applications. For example, when the filter is used to process flows of packets in a network and each flow can have potentially many packets but the number of packets of each flow is not known in advance. This opens an opportunity to adapt the filter when a false positive is detected so that subsequent lookups for the same element do not return a positive [14]. This can significantly reduce the number of false positives for workloads with repeated elements. The first practical implementation of adaptive filters was recently presented in [15] and extended cuckoo filters to support adaptation. The evaluation results demonstrated a large reduction in the number of false positives, for example when the filters are used to process network traffic. However, adaptive cuckoo filters have some limitations and drawbacks and thus it would be beneficial to have alternative schemes to implement adaptive filters.

To the best of our knowledge, so far no mechanism has been proposed to support adaptation in Bloom filters, reacting to false positives to eliminate them. In this paper, such a scheme

This work was supported by the ACHILLES project PID2019-104207RB-I00 and the Go2Edge network RED2018-102585-T funded by the Spanish Agencia Estatal de Investigación (AEI) 10.13039/501100011033 and by the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496.

[1]Pedro Reviriego and David Larrabeiti are with Universidad Carlos III de Madrid, Leganés 28911, Madrid, Spain. email: revirieg,dlarra@it.uc3m.es

[2]Alfonso Sánchez-Macian is with ARIES Research Center, Universidad Antonio de Nebrija, 28040, Madrid, Spain. email: asanchep@nebrija.es

[3]Ori Rottenstreich is with the Taub Department of Computer Science and the Viterbi Department of Electrical Engineering, Technion, Israel. email: or@technion.ac.il

is presented and evaluated. In more detail, the main benefits of the proposed scheme are:

1) To support adaptation in Bloom filters.
2) To implement adaptation with only one external memory access.
3) To reduce the false positive rate in practical scenarios, specially when the number of memory bits per element is low.
4) To complete all lookups in one memory access.

We evaluated the proposed scheme and the results show that adaptive Bloom filters can effectively reduce the number of false positives while completing all lookups in a single memory access. This makes them attractive for high-speed networking applications for which accessing the memory can be a bottleneck. Adaptive Bloom filters are of interest as they can provide better speed than cuckoo filters and achieve lower false positive rates for some settings.

The rest of the paper is organized as follows. Section II covers the preliminaries on Bloom filters and adaptation on filters. The proposed Adaptive Bloom Filters (ABFs) are presented in section III and analyzed theoretically in section IV. The performance of the ABFs is evaluated in section V both for synthetic workloads and for real packet traces to illustrate the benefits of the proposed adaptive Bloom filters. Section VI discusses how the proposed ABFs can be extended and generalized. Finally, the paper ends with the conclusion and ideas for future work in section VII.

II. Preliminaries

A Bloom filter is formed by a bit array of size m to which elements are mapped using k hash functions $g_1(x), \ldots, g_k(x)$ computed for an element x as shown in Figure 1. To insert an element x into the filter, positions $g_i(x)$ are set to one in the array. Conversely, to check if an element has been inserted into the filter, positions $g_i(x)$ are read and when they are all equal to one, a positive is obtained as a response. Otherwise, the element is for sure not in the filter and the response is negative. We shall call the elements not in the set of elements represented by the Bloom filter *negative elements*.

By construction, Bloom filters suffer from false positives as the positions an element maps to may have been set to one upon inserting other elements. Two parameters are commonly used to measure false positives: the *false positive probability* and the *false positive rate*. The first one gives the probability that a randomly chosen element not in the set yields a false positive. Instead, the second gives the fraction of false positives for a given collection of queries for negative elements. In expectation over uniformly selected elements, the false-positive rate should be equal to the false-positive probability. However, in many cases, the elements queried are not randomly chosen and the same elements may appear multiple times. In that case, if elements that yield false positives are queried many times, then the false-positive rate may be much higher than the false-positive probability.

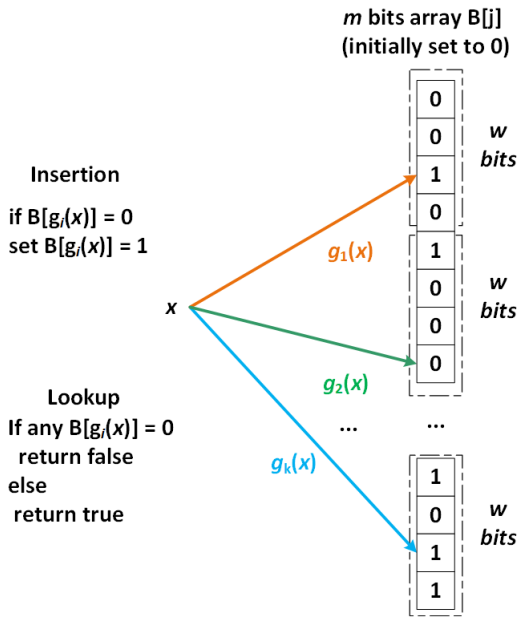

Fig. 1: Illustration of a Bloom filter. Each element is first mapped to k positions using hash functions $g_i(x)$ and those bits are set (insertion) or checked (lookup).

The False Positive Probability (FPP) of a Bloom filter depends on the number of elements stored in the filter n, its size m and the number of hash functions k. The FPP can be approximated by:

$$FPP \approx \left(1 - \left(\frac{m-1}{m}\right)^{n \cdot k}\right)^k. \quad (1)$$

When the size of the filter m is large, this value can be approximated by the simpler expression:

$$FPP \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k. \quad (2)$$

Instead, when m is small (e.g., less than a few hundred bits) more complex equations are needed to obtain an accurate estimate [16].

The traditional bit-based Bloom filters just described have some drawbacks when implemented in modern memory systems. The first one is that memories are in most cases larger than one bit wide. For example, most modern processors use words of 64 bits. This implies reading many additional bits that are not needed to check a single bit thus wasting power and memory bandwidth. A second issue is that in the worst case, an operation on the filter requires up to k memory accesses.

To address those issues, block Bloom filters were proposed in [17]. In a block Bloom filter, a first hash function $h(x)$ is used to select a block of bits, and then a group of hash functions $g_i(x)$ is used to select bits within that block. This ensures that the bits read are physically close and thus can be accessed faster, for example when the block size matches the cache line size. The concept of block Bloom filters was later optimized in [8] by mapping each block to exactly one memory word. This means that all query operations can

TABLE I: Summary of main notations

Symbol	Meaning
k	number of hash functions
M	number of words in the filter
w	number of bits per word
N	number of elements inserted in the filter
n_i	number of elements inserted in the i^{th} word of the filter
A	number of negative elements queried in the filter
a_i	number of negative elements queried in the i^{th} word of the filter
$h(x)$	word selection hash function
$g_i(x)$	first set of bit selection hash functions
$f_i(x)$	second set of bit selection hash functions
s	number of selector bits
S	number of sets of bit selection hash functions
d	decimation rate for adaptation
Z_i	number of elements that are false positive on the i^{th} set of bit selection hash functions
ABF	array of M words that stores the filter
$Bloom_{1G}$	array of M words that stores the external Bloom-1 filter for set $g_i(x)$
$Bloom_{1F}$	array of M words that stores the external Bloom-1 filter for set $f_i(x)$

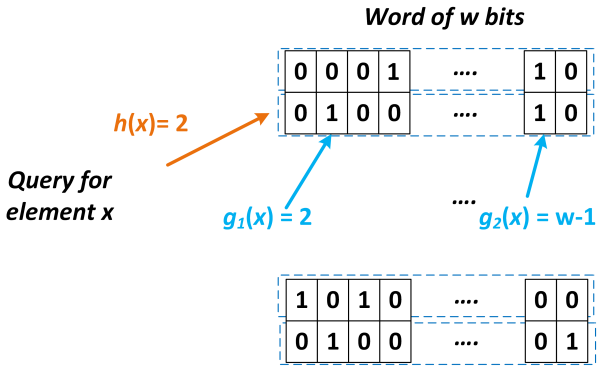

Fig. 2: Illustration of a Bloom-1 filter, each element is first mapped to a word with $h(x)$ and then $k = 2$ bits in indices $g_1(x), g_2(x)$ in that word are set (insertion) or checked (lookup).

be completed in one memory access. This filter, denoted as Bloom-1 is illustrated in Figure 2. The concept of word-based Bloom filters can also be extended by mapping to more than one word, to a bounded number of words. In more detail, word selection hash functions $h_1(x), \ldots, h_j(x)$ are used to select j words and in each, a set of bit selection hash functions is used. These are known as Bloom-j filters. However, in that case, some operations including positive lookups require j memory accesses, making it less attractive for high speed implementations.

In many applications, the access pattern is biased. This is, for example, the case in packet processing when elements are flows and each flow is queried for each of its packets. A key observation is that, in that case, the number of false positives can be reduced by detecting elements that are false positives and changing the filter to avoid further positives for those elements. This can be done, for example, by changing the fingerprint that creates a false positive for filters that rely on fingerprints [14]. The change or adaptation of the filter has to be simple so that its cost is negligible compared to the benefit of the reduction in false positive rate that it provides. This is not straightforward for Bloom filters as, for example, setting one of the bits to zero to remove a false positive can create false negatives [18]. Instead, in cuckoo filters, the fact that each element is associated with a fingerprint in the filter makes adaptation simpler as only that fingerprint needs to be changed and this can be done without creating false negatives for other elements. Indeed, the adaptive cuckoo filters [15] have shown how adaptation can be efficiently implemented in cuckoo filters. On the other hand, for Bloom filters, performing a simple filter modification to remove false positives without creating false negatives is not straightforward. In the next section, it is shown that, by using Bloom-1 word based filters, adaptation can be implemented efficiently in Bloom filters. The main notations used in the rest of the paper are summarized in Table I.

III. Adaptive One Memory Access Bloom Filters (ABFs)

This section first discusses the motivation and goals for the proposed filters and then presents and describes the Adaptive One Memory Access Bloom Filters.

A. Motivation and Goals

Many networking systems are implemented with ASICs [19] or FPGAs [20] on which there is a limited amount of on-chip fast memory and much a larger but slower external memory [21]. Therefore, we consider systems that have abundant slow off-chip memory but with limited bandwidth and a small but faster on-chip memory. In these systems, per packet operations need to be handled on the fast memory while the slow memory can be used for operations that are done on a small fraction of the packets only. The filters are commonly used to reduce the number of accesses to the external memory [22]. In fact, in many cases, the filter is an approximate representation of a full hash table that is stored in the external memory [15] so that on a positive the full table is checked. This means that false positives on the filter are detected when checking the full table as part of the packet processing. This facilitates adaptation as detecting false positives is needed to trigger

adaptation. Finally, on our target systems lookups are the most frequent operations as they are done per packet and a system can process many millions of packets per second. Instead, insertions or removals are orders of magnitude less frequent as they only occur when the state of the system changes, for example when a route is updated in BGP that occurs at relatively low rate [23]. A similar reasoning applies partly to packet processing systems that are implemented in servers with the use of processor caches and an external DRAM. In that case, the external DRAM is very large but has a high latency and limited bandwidth and the processor caches are much faster but smaller [22]. In that scenario, we would like all the lookups to complete on a single cache line access.

The design of the proposed adaptive one memory access Bloom filters has several goals:

- Reduce the false positive rate in scenarios where the memory is scarce so that only eight or fewer memory bits can be allocated per element.
- Complete all lookup operations in one memory access so that the maximum number of lookups can be executed for a given memory bandwidth.
- Have a design as simple as possible on which adaptation can be performed efficiently.

The combination of these features can make filters attractive, for example, in high-speed network applications that have to process hundreds of millions of packets per second posing a challenge in terms of memory bandwidth [22]. In fact, in packet processing, typically a few elements are queried many times and thus adaptation is expected to provide significant benefits as shown in [15].

B. Filter Design and Description

To achieve those goals, we propose to extend the Bloom-1 filter presented in [8] to support adaptation. The overall structure of the proposed adaptive Bloom filter is illustrated in Figure 3. To support adaptation while minimizing the adaptation cost, two groups of k hash functions: $g_i(x)$ and $f_i(x)$ are used to determine the bits in the word to set upon insertion (or to check upon a lookup). Then, two Bloom-1 filters are constructed using the same word selection hash function $h(x)$ and the two groups of bit selection hash functions $g_i(x)$,$f_i(x)$. These are stored in the external slower memory. The adaptive Bloom filter is located in the fast memory and for each word it uses the content of one of the external filters. The group of bit selection hash functions $g_i(x)$,$f_i(x)$ is determined by a selector bit that is added to each word of the adaptive Bloom filter. Then, all the bits in positions $g_i(x)$ (respectively, $f_i(x)$) in the word should be one to obtain a positive response when the selector bit is zero (respectively, one). It is important to note that the critical resource in the target implementation is the fast memory while the slow memory is abundant. This is indeed the case in networking ASICs for which fast on-chip memory is constrained as discussed before [19].

Adaptation is performed when a false positive is observed upon the query of some element x. This is the case when the filter returns a positive indication but the element cannot be found in the checking of the full table. Then, the selector bit in word $h(x)$ is read. If the bit has a value of zero, word $h(x)$ from the external Bloom-1 filter with functions $f_i(x)$ is queried to check if x is a negative. If so, the contents of the Bloom-1 filter with functions $f_i(x)$ for that word are written in the adaptive Bloom filter to replace word $h(x)$, setting the selector bit to one. Then, if element x is queried again, it will return a negative. Instead, when x returns a positive on the external Bloom-1 filter with functions $f_i(x)$, the false positive cannot be removed. This however will occur with low probability as the false positive probability of each Bloom-1 is low. Similarly, if the selector bit was set to one, the check would be done on the external Bloom-1 filter with functions $g_i(x)$. As can be observed, the adaptation process is simple and requires only one access to the external memory. For applications on which the full table is not stored on the external memory, it is also possible to detect most false positives by checking the alternative Bloom-1 on the external memory on a positive. If a negative is obtained on the alternative Bloom-1, a false positive is detected. To limit the amount of external memory accesses this can be done for a fraction of the positives only.

The selector bits are initialized to zero and the word contents to those of the first set of hash functions $g_i(x)$. The operations on the adaptive Bloom filter are more formally described in Algorithms 1, 2 and 3. It can be seen that the lookup or check only requires to access one word and thus only one memory access. Instead, insertions and adaptations are slightly more complex. This should not be an issue as the most frequent operation of the filters in our target systems are lookups as discussed before.

The lookup is described in Algorithm 1. It basically reads the corresponding word in the ABF, extracts the selector bit and based on its value applies one of the groups of bit selection hash functions to determine the bits to check. Compared with the lookup in a Bloom-1 filter, the complexity is similar and the number of memory accesses is the same, just one.

Algorithm 1 Procedure ABF: Element Query x

```
W = ABF[h(x)].
if W[selectorbit] == 0 then
    for i = 1 to k do
        if W[g_i(x)] == 0 then
            return negative;
        end if
    end for
    return positive;
else
    for i = 1 to k do
        if W[f_i(x)] == 0 then
            return negative;
        end if
    end for
    return positive;
end if
```

Algorithm 2 describes the insertion procedure. In this case,

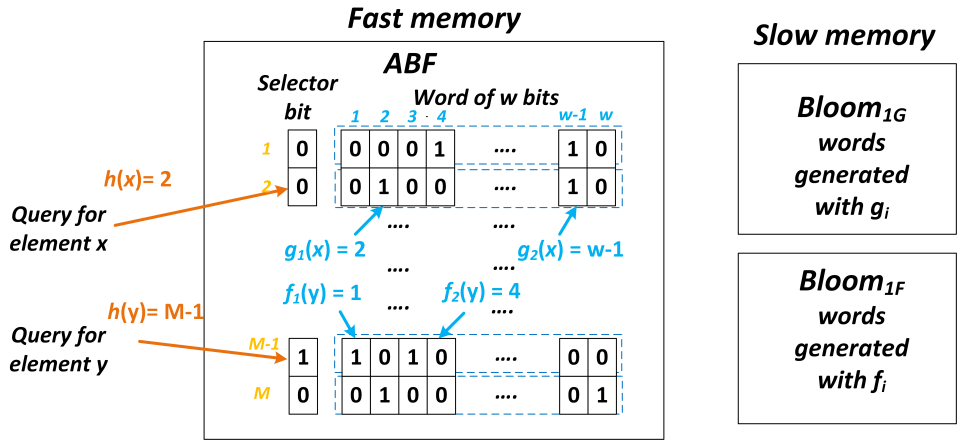

Fig. 3: Illustration of the proposed adaptive Bloom filter. Each word has a bit to select the hash functions to use. When the selector is zero use $g_i(x)$ and when it is one use $f_i(x)$.

in addition to the adaptive Bloom filter, both external Bloom-1 filters need to be updated requiring two additional external memory accesses for read and write. As insertions are typically orders of magnitude less frequent than lookups in networking applications [23], the impact of those additional accesses on the memory bandwidth would be negligible.

Algorithm 2 Procedure ABF: Element Insertion x

$W = ABF[h(x)]$.
$WG = Bloom_{1G}[h(x)]$.
$WF = Bloom_{1F}[h(x)]$.
for $i = 1$ to k **do**
  if $W[selectorbit] == 0$ **then**
    $W[g_i(x)] = 1$;
  else
    $W[f_i(x)] = 1$;
  end if
  $WG[g_i(x)] = 1$;
  $WF[f_i(x)] = 1$;
end for
$ABF[h(x)] = W$.
$Bloom_{1G}[h(x)] = WG$.
$Bloom_{1F}[h(x)] = WF$.

Finally, the adaptation procedure is presented in Algorithm 3. First, the word of the filter that has suffered the false positive, $ABF[h(x)]$ is read. If the selector bit is zero (one), that means that the first (second) set of bit selection hash functions $g_i(x)$ ($f_i(x)$) is being used so adaptation is tried with the second (first) set $f_i(x)$ ($g_i(x)$). To do so, the relevant word $Bloom_{1F}[h(x)]$ ($Bloom_{1G}[h(x)]$) is read and we check that x is a negative on that word. If that is the case, then word $ABF[h(x)]$ in the filter is written with the corresponding word $Bloom_{1F}[h(x)]$ ($Bloom_{1G}[h(x)]$) and the selector bit is set accordingly. In this case, only one external memory read access is needed plus a write to the adaptive Bloom filter when adaptation succeeds. As adaptation is only triggered when a false positive occurs and those should be a small fraction of the lookups in practical configurations, the overhead introduced by adaptations should be low.

Algorithm 3 Procedure ABF: Element Adaptation x

$W = ABF[h(x)]$.
if $W[selectorbit] == 0$ **then**
  $WF = Bloom_{1F}[h(x)]$.
  for $i = 1$ to k **do**
    if $WF[f_i(x)] == 0$ **then**
      $ABF[h(x)] = WF$ with selector bit = 1.
      return true;
    end if
  end for
  return false;
else
  $WG = Bloom_{1G}[h(x)]$.
  for $i = 1$ to k **do**
    if $WG[g_i(x)] == 0$ **then**
      $ABF[h(x)] = WG$ with selector bit = 0.
      return true;
    end if
  end for
  return false;
end if

The adaptation capability of the proposed adaptive Bloom filter can be improved by using more than two sets of bit selecting hash functions. For example, two selector bits can be used so one of four sets of hash functions can be chosen to eliminate false positives. In general s selector bits can be used to support up to $S = 2^s$ sets of bit selection hash functions. The extension of the algorithms for insertion, lookup, and adaptation presented for $s = 1$ to the general case is straightforward. This extension requires s selection bits per word on the adaptive Bloom filter and 2^s Bloom-1 filters in the slower memory. Therefore, it should only be used when it reduces the false positive rate significantly over the use of two sets. In section V, adaptive Bloom filters with more than two sets of bit selecting hash functions will be evaluated to show when they provide significant benefits.

Finally, the use of the s selector bits reduces the fast memory available for the filter by a ratio $\frac{s}{w} < 1$. For example, for $w = 64$ and $s = 1, 2, 3$ the memory is reduced by

approximately 1.6%, 3.1% and 4.7% respectively. The impact of those reductions on the false positive rate are negligible compared to the reduction on the false positive rate that the ABF provides for skewed traffic as will be seen in the experimental evaluation (Section V). Indeed, reducing the false positive rate by a factor of two already requires 1.44 bits[1] per element stored in the filter and typically the number of elements is at least eight.

IV. Analysis

In this section, analytical expressions for the false positive rate of the proposed adaptive Bloom filters are derived for the case of two sets of bit selection hash functions when the elements appear with the same probability. The extension of the expressions for a larger number of sets of bit selection hash functions is straightforward.

Before starting the analysis it is worth to discuss a few examples to illustrate how adaptation can reduce the false positive rate. Let us consider two elements x_1, x_2 that map to the same block of the ABF being false positives on the g_i and f_i groups of bits selection hash functions respectively. Then if the elements are queried multiple times, the ABF would adapt back and forth from one group of hash functions to the other. It may seem that this would result in a larger number of false positives. Let us consider the following sequence of queries $x_1, x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_1, x_1, x_1$. Here, a non adaptive filter would suffer ten false positives, one per each query to x_1. The ABF would adapt on the first query, then again on the query for x_2 and a third time on the following query for x_1, so having only three false positives.

This illustrates how even in this scenario on which we have false positive elements on both groups of bits selection hash functions, the ABF can exploit the skew in the number of queries per element to reduce the false positive rate. This is of interest in networking applications as traffic tends to be highly skewed. Finally, the worst case would be an element x_1 that is a false positive in both groups of bits selection hash functions. In this case the ABF would not be able to reduce the false positives, however these would be only a small fraction of the original false positives.

Let us start by considering the false positive rate for one of the words in the proposed adaptive Bloom filter. In more detail, let us assume that the word has w bits (in addition to the selector bit), that k bit selection functions are used, and that n elements have been inserted on the word. Since w is small, the traditional approximation for the false positive rate is not accurate and more accurate expressions have to be used. In particular, the false positive rate $f(n, w)$ can be estimated using the following formula presented in [16]:

$$f(n,w) \approx \frac{w!}{w^{k\cdot(n+1)}} \cdot \sum_{i=1}^{w}\sum_{j=1}^{i}(-1)^{i-j}\frac{j^{kn}\cdot i^k}{(w-i)!j!(i-j)!} \quad (3)$$

[1]This is the number of bits needed to reduce the false positive rate of a Bloom filter by 2x in its optimal configuration [3].

Adaptation will be used only when the first set of bit selecting hash functions $g_i(x)$ exhibit false positives. Following [15], the number of false positives on a word to which a negative elements are queried can be modeled as a binomial distribution with a tries each with probability of success $f(n, w)$, whose probability distribution function is denoted in the following as $b(a, f, x)$. Therefore, the probability that the first set of bit selecting hash functions has false positives can be approximated by $1 - b(a, f(n, w), 0)$. When that occurs, there are two possibilities: 1) the second set of hash functions has no false positives and 2) the second set of hash functions does have false positives. In the first case, after the first positive adaptation is performed no further positives occur. Instead, in the second case, the word will keep adapting between the two sets of hash functions.

The first case occurs with a probability approximated as:

$$P_1 \approx (1 - b_a) \cdot b_a \quad (4)$$

for $b_a = b(a, f(n, w), 0)$.

Assuming for now that all a elements appear $t \geq 1$ times, in this scenario the word will experience a false positive rate of $F_1 = \frac{1}{a\cdot t}$ compared to at least $\frac{1}{a}$ of a traditional Bloom filter.

For the second case, the number of elements that create false positives on each set of hash functions are denoted as $Z_1 > 0$ and $Z_2 > 0$, respectively. Then, the probability that Z_1, Z_2 false positives are observed on each configuration can be approximated by:

$$P_2(Z_1, Z_2) = b(a, f(n, w), Z_1) \cdot b(a, f(n, w), Z_2) \quad (5)$$

In that scenario, the word will loop between the two sets of hash functions as false positives occur. Therefore, it will stay in the first state on average $\frac{a}{Z_1}$ lookups and on the second state $\frac{a}{Z_2}$ before going back to the first state. This gives a false positive rate of:

$$F_2(Z_1, Z_2) = \frac{2}{\frac{a}{Z_1} + \frac{a}{Z_2}} \quad (6)$$

It can be seen that the false positive rate will be low when either Z_1 or Z_2 are small.

In deriving this equation, it has been assumed that all the a elements appear with the same probability. In many applications, that is not the case and the frequency of elements follows a heavy tailed distribution like for example a Pareto or a Zipf distribution [14]. In that case, the fraction of false positives when Z_1 elements are false positives would be given by the sum of the frequencies of those Z_1 elements. This corresponds to a sum of heavy tailed distributions that has been shown to still have a skewed distribution [24]. This would tend to make adaptation more effective as will be seen in the evaluation results for packet traces that are heavily skewed presented in the next section.

Finally, combining the previous equations, the false positive rate for the word can be estimated by:

$$F \approx P_1 \cdot F_1 + \sum_{Z_1, Z_2 > 0} P_2(Z_1, Z_2) \cdot F_2(Z_1, Z_2). \quad (7)$$

The first term corresponds to the first case on which after the first adaptation there are no further false positives. The second term corresponds to the case on which there are false positives on both states and adaptation keeps transitioning between the two states. The adaptation will be able to reduce the false positive rate when Z_1 is larger than Z_2. Instead when Z_2 is larger than Z_1 adaptation would increase the false positive rate. However, on average adaptation would reduce the false positive rate as the benefits of the first case are larger than the losses introduced by the second case. This is due to the harmonic averaging done in equation (6). Therefore, the larger the differences between Z_1 and Z_2, the greater the benefits of adaptation. These differences will tend to be larger when a is small. For the more general case of non uniform element frequencies, the differences can still be significant when a is large as the sum of heavy tailed distributions tends to still be skewed as discussed before.

To obtain the false positive rate for the entire filter with M words, the false positive rates of all its words need to be combined. Those false positive rates would be in general different as the number of elements stored (queried) to each word n_i (a_i) is different. As discussed in [8], for a filter with M words that store N elements, the probability of storing n_i elements is given by:

$$P(n_i) = \binom{N}{n_i} \cdot \left(\frac{1}{M}\right)^{n_i} \cdot \left(\frac{1}{M}\right)^{N-n_i} = \binom{N}{n_i} \cdot \left(\frac{1}{M}\right)^{N} \quad (8)$$

The same expression can be used for the probability of mapping a_i elements to a word when A elements are queried in the entire filter:

$$P(a_i) = \binom{A}{a_i} \cdot \left(\frac{1}{M}\right)^{a_i} \cdot \left(\frac{1}{M}\right)^{A-n_i} = \binom{A}{a_i} \cdot \left(\frac{1}{M}\right)^{A} \quad (9)$$

Finally, putting all together and taking into account that the weight of each word towards the false positive rate depends on the ratio of a_i to the average number of negatives per word $\frac{A}{M}$ we obtain:

$$FPR \approx \sum_{n_i, a_i} P(n_i) \cdot \frac{a_i \cdot M}{A} \cdot P(a_i) \cdot F(n_i, a_i) \quad (10)$$

This expression approximates the average false positive rate of a filter. In deriving the equation, it has been assumed that the false positive probability of a w bit word that stores n elements is given by $f(n, w)$. This is the expected probability but when w is small as in our case, each instance of such a word will have a probability that can deviate from that value. That would benefit adaptation as it will tend to stay more time in the word instance that has the lowest false positive probability. Therefore, the approximation given by equation (10) tends to be larger than the average false positive rate of the filter. Finally, since in practical settings, filters are composed of many words, the observed false positive rate should be close to the average. To extend the approximation to the general case of S sets of bit selection hash functions, the same derivation can be used but having 2^s states instead of two states on which we have Z_i elements that have false positives.

A potential issue of the proposed adaptive Bloom filters is the cost of adaptations. Each adaptation requires to access the alternative Bloom filter word on the external memory and write it on the fast memory. Let us consider our target system in which the filter is used to reduce the number of accesses to the full table of elements that is stored on the external memory. Then, a false positive would typically require several external memory accesses depending on the type of hash table used to store the elements. Additionally, checking each element would also consume more memory bandwidth than reading a Bloom filter block. For example, when elements are IPv4 5-tuples they have more than 100 bits and close to 300 for IPv6 so much larger than a 64 bit Bloom filter word. Therefore, the external memory access for adaptation would have a much lower cost than that of a false positive. Additionally, the previous analysis suggests that when t is sufficiently large, the false positive rate could be similar if adaptation is done only each d false positives instead of on each false positive. In more detail, in equation (7), only the first term would change when adapting each d false positives while the second would remain the same. That is, when there are false positives in both states, the adaptation speed should not impact the false positive rate because regardless of the adaptation speed, the filter will stay on each state the same fraction of the time. Instead for the case of no false positives in the second state, the false positive rate would increase by a factor of d. Therefore, the impact of reducing the adaptation speed would depend on the number of times that elements appear and on the relative importance of the two terms of equation (7). In the next section, this will be evaluated in one of the simulation experiments. Finally, it is important to bear in mind that adaptations are only done when false positives occur and thus would be much less frequent than lookups in all cases.

V. Experimental Evaluation

The proposed adaptive Bloom filters have been implemented in Python[2]. For simplicity, in the implementation, adaptation is always done without checking if the new set of bit selecting hash functions eliminates the false positive. This should have a minor impact as the false positive rate is low and thus most adaptations would indeed eliminate the false positive. In any case, checking if adaptations remove the false positive would result in better performance for the proposed filters when S is larger than two, so the results presented in the following are actually a worst case. To compare filters that use exactly the same amount of fast memory, the size of the filter is set to w

[2]The code is available at https://github.com/amacian/ABF

bits for the Bloom-1 filter and to $w-s$ for the ABF. As will be shown later despite this small reduction in filter size, the ABF is still able to reduce the false positive rate with adaptation.

The code has been used to evaluate the performance of the proposed adaptive Bloom filters in different scenarios. The main goals are to first check that the result match the analytical expressions obtained in section IV for synthetic workloads and then to evaluate the benefits of the proposed filters in a real use case. In the first set of experiments, synthetic workloads are used and the simulation results are compared with those of the theoretical model presented in section IV. Then, in a second set of experiments, the performance of the filters is evaluated using real packet traces to illustrate the potential benefits of the proposed filter in practical scenarios. Since the proposed filters complete all lookups in one memory access, the main reference for comparison are Bloom-1 filters that are the only ones that provide the same feature. Additionally, comparisons are also made with adaptive cuckoo filters to assess the benefits of adaptation. In all the comparisons, the same amount of fast memory is allocated to the different filters considered to make the comparison fair.

A. Synthetic workloads

To evaluate the proposed filters with synthetic workloads, first, N elements are generated and inserted in the filter. Then, A negative elements are generated and finally, $A \cdot t$ lookups are performed taking one element randomly from the A negative elements with all having the same probability of being selected. This emulates a workload on which elements are repeated and all have the same probability of occurring. This is the same as assumed in the derivation of the theoretical model. The filter parameters are $M = 1024$ words each having $w = 64 - s$ bits and the results are the average over 10 runs.

In the first experiment, $N = 8 \cdot M$ elements are inserted in the filter corresponding to $\frac{64}{8}$, so eight bits per element. The number of negatives is set to $A = N, 2N, 3N, ..., 10N$ giving ratios of $\frac{A}{N} = 1, 2, 3..., 10$. Finally, values of $t = 10$ and $k = 3, 4, 5$ were tested. The results are shown in Figure 4. The value of k that provides the lowest false positive rate is $k = 4$ which is lower than the one for a traditional Bloom filter, this is due to the fact that in a Bloom-1 filter, words to which many elements map tend to degrade the false positive rate of the entire filter and a lower k reduces the false positive rate of those words. It can be observed that the proposed filters are only able to reduce the false positive rate when the $\frac{A}{N}$ ratio is small. In fact, the false positive rate for both the traditional Bloom-1 and the proposed ABF are significantly larger than those of the cuckoo filter and the adaptive cuckoo filter respectively [15]. This can be expected as the traditional Bloom filter has a larger false positive probability than the cuckoo filter in this configuration. In fact, the Bloom-1 has worse false positive rate than traditional Bloom filters in exchange for completing all lookups in a single memory access. Finally, the theoretical and simulation results are very similar in all cases which suggests that the theoretical model can be used to estimate the false positive rate of adaptive Bloom filters when all elements appear the same number of times.

As discussed in [4], the cuckoo filter cannot reduce the number of fingerprint bits below six without impacting the maximum occupancy that can be achieved and thus it is of interest to evaluate the proposed filters for lower bits per element. In the second experiment, the number of elements in the filter was increased to $N = 12 \cdot M$, so fewer than 6 bits per element and the same configurations were tested. The results are summarized in Figure 5. It can be seen that the proposed adaptive Bloom filters are still able to reduce the false positive rate for low $\frac{A}{N}$ ratios. This suggests that adaptive Bloom filters can be useful when the number of memory bits per elements is low.

B. Packet traces

The synthetic workloads used in the first experiments are actually a worst case of repetitions for adaptive filters as noted in [15]. In many applications, the frequency of the elements is different, and a few elements account for many of the lookups. This is, for example, the case in network traffic where a few flows have many packets and the rest have just a few [25]. Intuitively, this is better for adaptation as false positives on flows with many packets can be removed by adaptation. The performance of the proposed adaptive Bloom filter has been evaluated using packet traces from high speed Internet links[3]. The traces correspond to one minute of traffic containing millions of packets and the number of packets per flow is highly skewed. This is shown in Figure 6 that presents the number of flows having 1-10,11-100,101-1000 and 1001-10000 packets in each of the traces. It can be observed that most of the flows have only a few packets but there are a few that have many packets. Removing false positives on those flows with the high number of packets is the focus of the suggested adaptation.

In the simulations, for each trace, N randomly selected elements have been added to the filter and the false positive rate when performing a lookup for each packet in the trace has been measured. This would correspond to the sampling of N flows on the link. The experiment has been done for $N = 8M, 12M, 16M$ that correspond approximately to 8,5 and 4 bits per element and different values of k. Ten different random selections of the N elements inserted in the filter have been simulated and the average value is reported. In theses experiments, different values of S were also tested to evaluate the potential benefits of using more than two sets of bit selection hash functions.

The results are presented in Figures 7, 8, 9 for $N = 8M, 12M, 16M$. The simulation results for the Bloom-1 are also included as well as the theoretical estimate of their false

[3]The same traces in [15] have been used to make results directly comparable. Those correspond to CAIDA 2014 dataset [26] and are equinix-sanjose.dirA.20140320-130400 (Trace 1) equinix-chicago.dirA.20140619-130900 (Trace 2) and equinix-chicago.dirB.20140619-132600 (Trace 3).

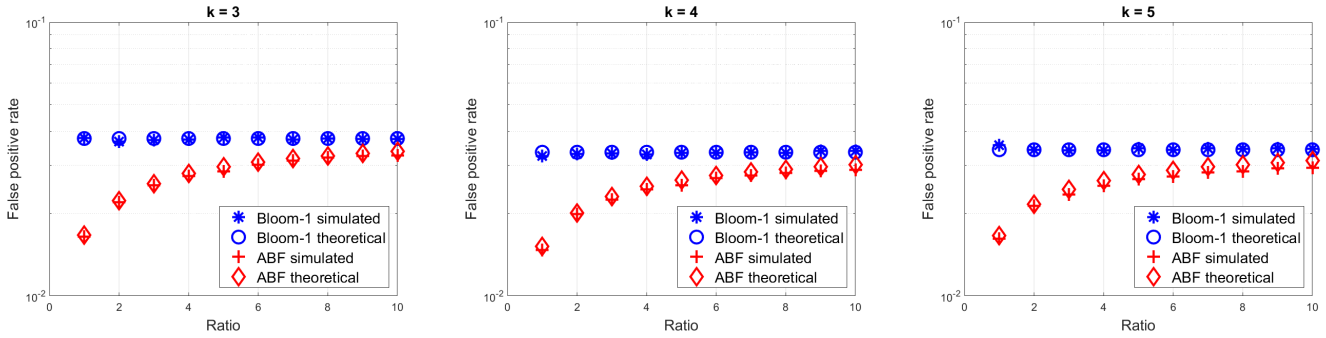

Fig. 4: False positive rate versus A/N ratio when $N = 8 \cdot M$ elements are stored in the filter for $k = 3$ (left), $k = 4$ (middle), and $k = 5$ (right).

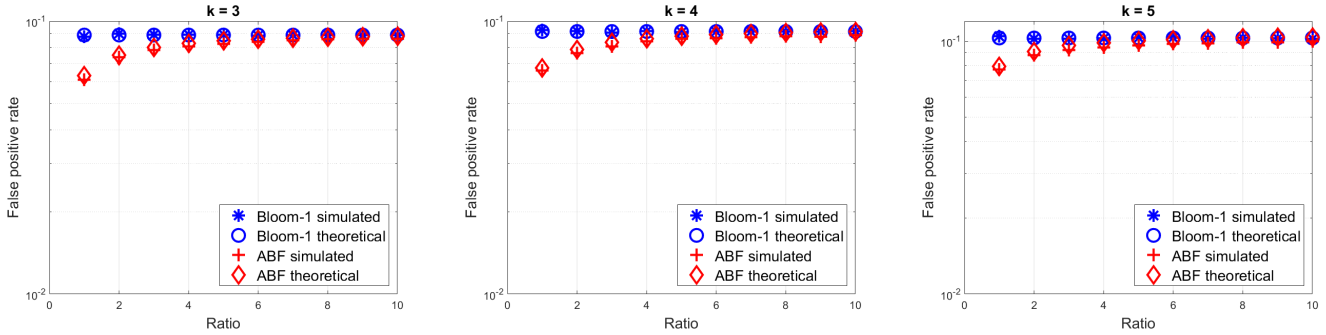

Fig. 5: False positive rate versus A/N ratio when $N = 12 \cdot M$ elements are stored in the filter for $k = 3$ (left), $k = 4$ (middle), and $k = 5$ (right).

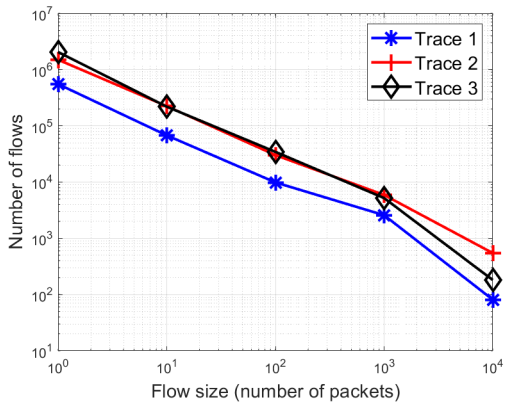

Fig. 6: Number of flows on the packet traces with a number of packets in a given interval, 1-10,11-100,101-1000 and 1001-10000 (the x-axis value in the plot corresponds to the maximum value in the interval).

positive rate provided in [27]. It can be observed that for the traditional Bloom-1, the simulation results match the theoretical estimate. Comparing the proposed adaptive Bloom filters with the traditional Bloom-1, it can be seen that they reduce the false positive rate for all traces in all the configurations. The lowest false positive rates are obtained when using $S = 8$ sets of bit selection hash functions but the benefit over using $S = 4$ is small and in some cases negligible. However, as discussed before, larger values of S require additional Bloom-1 filters in the external memory and selector bits in the adaptive Bloom filter. Therefore, it seems that $S = 4$ would be the best option when filters are used to process network traffic.

The simulation results are summarized in Table II. For each filter, the k that gives the lowest false positive rate is reported. This enables a direct comparison of the different filters and shows the benefits of the proposed scheme. In more detail, the reductions in false positive rate are at least 2.02x, 2.67x, 2.77x for $S = 2, 4, 8$. The lowest false positive rates are achieved when $S = 8$ but the benefit over $S = 4$ is limited. The reductions are larger when $\frac{N}{M}$ is smaller corresponding to larger number of memory bits per element. There is some variation on the results with the traces but the qualitative trends with S and $\frac{N}{M}$ are the same for all traces. Finally, comparing with the adaptive cuckoo filter [15], the reduction on the false positive rate is smaller. However, as discussed before, the adaptive cuckoo filter is not attractive when the number of bits per element is small. In summary, the proposed adaptive Bloom filters can effectively reduce the false positive for network traffic even when the number of memory bits per element is small.

The impact of the overhead introduced by adaptation depends on the relative cost of a false positive versus an adaptation that is system dependent as discussed in the previous section. In many cases, adaptation would require only a fraction of the cost of a false positive and thus the impact would be small. Additionally, for configurations in which the cost of an adaptation is similar to that of a false positive, the

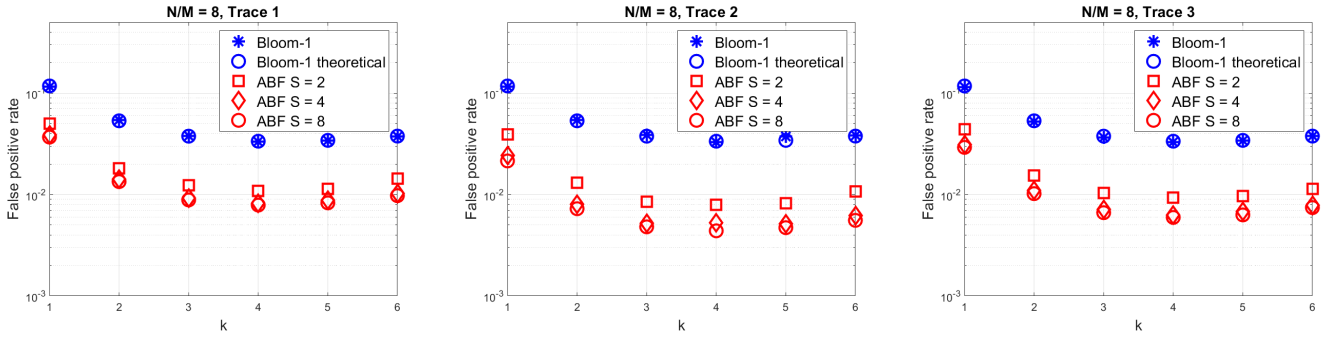

Fig. 7: False positive rate versus the number of hash functions k when $N = 8 \cdot M$ elements are stored in the filter for Trace 1 (left), Trace 2 (middle), and Trace 3 (right).

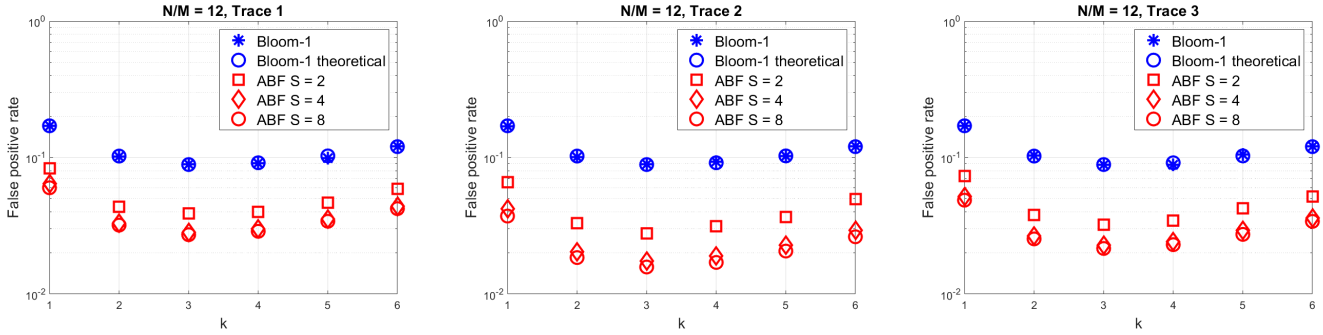

Fig. 8: False positive rate versus the number of hash functions k when $N = 12 \cdot M$ elements are stored in the filter for Trace 1 (left), Trace 2 (middle), and Trace 3 (right).

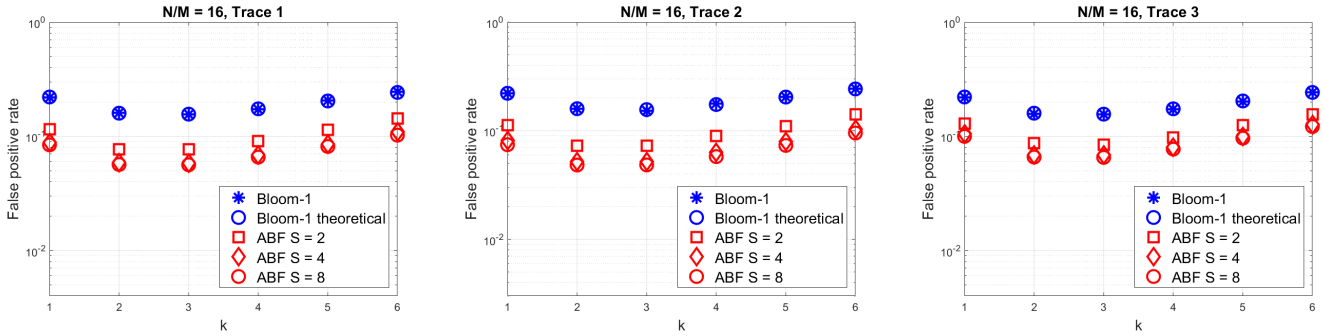

Fig. 9: False positive rate versus the number of hash functions k when $N = 16 \cdot M$ elements are stored in the filter for Trace 1 (left), Trace 2 (middle), and Trace 3 (right).

adaptation speed can be reduced so that the overhead is small. In the last experiment, the impact of reducing the adaptation speed on the false positive rate was explored. To that end, the same simulations were run but instead of performing adaptation on each false positive, adaptation was done only once after 5 false positives. From the analysis presented in the previous selection, if flows have a large number of packets, this should not affect the effectiveness of the proposed scheme when the two states have false positives. In that case it only slows down transitions between the two states but does not change the probability of being in each state and thus the false positive rate does not change. In our case, the average number of packets per flow is not so large, partly due to using only one minute of traffic. Instead, if the second state has no false positives, reducing adaptation would increase the false positive rate by the decimation factor. Therefore, some degradation of the false positive rate is expected. The results for the first trace are presented in Figure 10 that shows the false positive rate for both no decimation and $d = 5$. It can be seen that there is indeed an increase on the false positive rate but even with decimated adaptation, the proposed filters are still able to significantly reduce the false positive rate compared to a traditional Bloom-1 filter.The results for the other two traces were similar showing reductions in the false positive rates. Therefore, for systems in which the cost of adaptation is large, the proposed adaptive filters can still be used with low overhead by reducing the adaptation speed.

TABLE II: Comparison of false positive rate of filters with different S for the k that achieves the lowest false positive rate

Trace	$\frac{N}{M}$	Bloom-1	ABF,$S=2$	Reduction	ABF,$S=4$	Reduction	ABF,$S=8$	Reduction	ACF	Reduction
Trace 1	8	0.0331	0.0109	3.04x	0.0082	4.03x	0.0079	4.19x	0.0019	17.42x
Trace 1	12	0.0894	0.0389	2.30x	0.0282	3.17x	0.0272	3.29x	N.A.	N.A.
Trace 1	16	0.1557	0.07771	2.02x	0.0582	2.67x	0.0563	2.77x	N.A.	N.A.
Trace 2	8	0.0328	0.0079	4.15x	0.0052	6.30x	0.0044	7.45x	0.0020	17.0x
Trace 2	12	0.0893	0.0278	3.21x	0.0174	5.13x	0.0158	5.65x	N.A.	N.A.
Trace 2	16	0.1533	0.0608	2.51x	0.0386	3.97x	0.0352	4.36x	N.A.	N.A.
Trace 3	8	0.0330	0.0093	3.55x	0.0063	5.24x	0.0059	5.59x	0.0024	13.66x
Trace 3	12	0.0885	0.0321	2.76x	0.0227	3.90x	0.0216	4.10x	N.A.	N.A.
Trace 3	16	0.1553	0.0680	2.28x	0.0481	3.23x	0.0458	3.39x	N.A.	N.A.

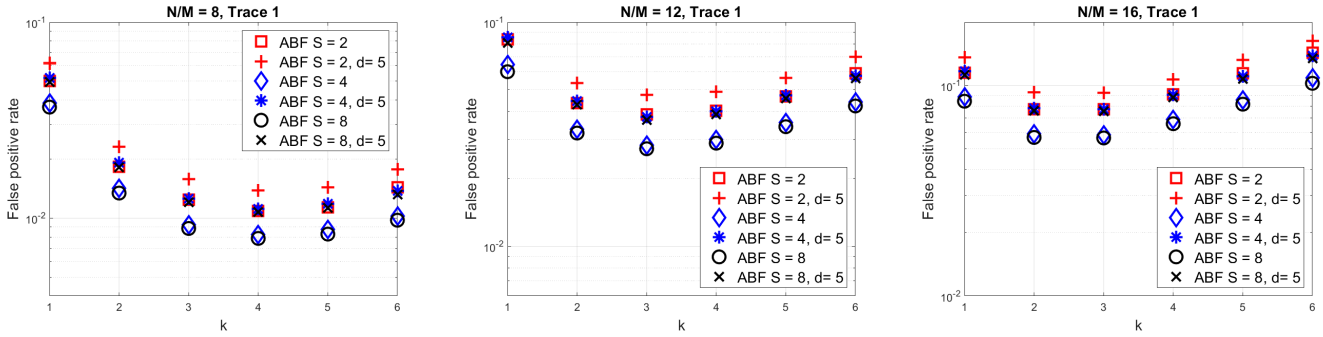

Fig. 10: positive rate versus the number of hash functions k with adaptation on every false positive $d = 1$ and decimated adaptation $d = 5$ for Trace 1 and $N = 8 \cdot M$ (left), $N = 12 \cdot M$ (middle), and $N = 16 \cdot M$ (right).

VI. Discussion and Generalization

In this paper we have focused on implementing adaptation on one memory access Bloom filters that are stored in fast memory having a larger slower memory on which additional information can be stored at low cost. However, the ideas presented can be extended in several directions. For example, the proposed adaptive Bloom filter can also be extended to general Bloom filters either word or bit based. For word based Bloom filters, the same scheme can be used keeping two different Bloom-j filters in the slower memory and performing adaptation on each of the j words until the false positive is removed. Instead, for traditional bit based Bloom filters that use a single array of bits, the array can be arranged in blocks of w bits that are treated as a word in Bloom-1 with a single hash function to select a bit. Then adaptation can be implemented by adding a selector bit per block and using similar procedures to those described described in section III.

Finally, in some implementations both the adaptive Bloom filter and the Bloom-1 filters may be stored in the same memory and the adaptive Bloom filter is used to reduce the number of memory accesses. In those scenarios, a potential optimization is to keep only one Bloom-1 filter that stores the word that corresponds to the bit selection hash functions not used for that word in the adaptive Bloom filter. That is, if the adaptive Bloom filter on a given word stores the configuration that corresponds to $g_i(x)$, then the Bloom-1 stores in that same word the configuration that corresponds to $f_i(x)$. This reduces the amount of memory needed at the cost of requiring an additional write access in adaptations to save the configuration being removed from the adaptive filter to the Bloom-1 so that it can be later used should more adaptations be needed.

VII. Conclusions and Future Work

In this paper, a scheme to implement adaptation in Bloom filters has been proposed and evaluated. The proposed design uses word-based Bloom filters and, on a false positive on a given word, it is replaced with the word from a second filter to remove it. This makes adaptation simple, requiring only one external memory access to retrieve the alternative word. Lookups in the filter are also always completed in a single memory access. This makes the proposed filters attractive for high-speed implementations. The evaluation results show that the proposed adaptive Bloom filters can significantly reduce the false positive rate in practical applications, especially when the distribution of the frequency of elements is skewed. From a different perspective, adaptation can be used to reduce the fast memory needed to achieve a target false positive rate.

The proposed scheme can also be used to implement adaptation in Bloom-j word-based filters and also in traditional Bloom filters by replacing blocks of bits to implement adaptation. Although those Bloom filters require several memory accesses for some lookups, they have lower false positive probabilities than Bloom-1. Therefore, their adaptive versions may also achieve lower false positive rates than the proposed one memory access adaptive filters. Analyzing and evaluating those extensions is an interesting area for future work. Finally, security is an increasingly important consideration for probabilistic data structures and thus studying the proposed filters in adversarial environments to see if they expose any new vulnerability is also a direction for future work.

REFERENCES

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, 1970.

[2] 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, 2012.

[3] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

[4] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *ACM CoNEXT*, 2014.

[5] L. Luo, D. Guo, O. Rottenstreich, R. T. B. Ma, X. Luo, and B. Ren, "A capacity-elastic cuckoo filter design for dynamic set representation," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 18, no. 4, pp. 4860–4874, 2021.

[6] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarelli, "Cuckoo filters and Bloom filters: Comparison and application to packet classification," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 17, no. 4, pp. 2690–2701, 2020.

[7] H. Lang, T. Neumann, A. Kemper, and P. Boncz, "Performance-optimal filtering: Bloom overtakes cuckoo at high throughput," *VLDB Endow.*, vol. 12, no. 5, p. 502–515, 2019.

[8] Y. Qiao, T. Li, and S. Chen, "Fast Bloom filters and their generalization," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 1, pp. 93–103, 2014.

[9] S. Z. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, "Bloom filter with a false positive free zone," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 18, no. 2, pp. 2334–2349, 2021.

[10] O. Rottenstreich, P. Reviriego, E. Porat, and S. Muthukrishnan, "Avoiding flow size overestimation in Count-Min sketch with Bloom filter constructions," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 18, no. 3, pp. 3662–3676, 2021.

[11] R. Durner and W. Kellerer, "Network function offloading through classification of elephant flows," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 17, no. 2, pp. 807–820, 2020.

[12] J. Bruck, J. Gao, and A. Jiang, "Weighted Bloom filter," in *IEEE International Symposium on Information Theory (ISIT)*, 2006.

[13] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing data popularity conscious Bloom filters," in *ACM symposium on Principles of distributed computing (PODC)*, 2008.

[14] M. A. Bender, M. Farach-Colton, M. Goswami, R. Johnson, S. McCauley, and S. Singh, "Bloom filters, adaptivity, and the dictionary problem," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2018.

[15] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, pp. 1–20, 2020.

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

[17] U. Manber and S. Wu, "An algorithm for approximate membership checking with application to password security," *Inf. Process. Lett.*, vol. 50, no. 4, p. 191–197, 1994.

[18] B. Donnet, B. Baynat, and T. Friedman, "Retouched Bloom filters: Allowing networked applications to trade off selected false positives against false negatives," in *ACM CoNEXT*, 2006.

[19] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM*, 2013.

[20] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[21] Y. Kanizo, D. Hay, and I. Keslassy, "Maximizing the throughput of hash tables in network devices with combined SRAM/DRAM memory," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 3, pp. 796–809, 2015.

[22] S. Pontarelli, P. Reviriego, and M. Mitzenmacher, "Emoma: Exact match in one memory access," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 30, no. 11, pp. 2120–2133, 2018.

[23] A. Elmokashfi, A. Kvalbein, and C. Dovrolis, "On the scalability of BGP: The roles of topology growth and update rate-limiting," in *ACM CoNEXT*, 2008.

[24] C. M. Ramsay, "The distribution of sums of i.i.d. pareto random variables with arbitrary shape parameter," *Communications in Statistics - Theory and Methods*, vol. 37, no. 14, pp. 2177–2184, 2008.

[25] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf's law for traffic offloading," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 1, p. 16–22, 2012.

[26] CAIDA. (2014) realtime passive network monitors. [Online]. Available: http://www.caida.org/data/realtime/passive.

[27] P. Reviriego, K. Christensen, and J. A. Maestro, "A comment on "fast Bloom filters and their generalization"," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 1, pp. 303–304, 2016.