Concise Papers

Aging Bloom Filter with Two Active Buffers for Dynamic Sets

MyungKeun Yoon, *Member*, *IEEE*

Abstract—A Bloom filter is a simple but powerful data structure that can check membership to a static set. As Bloom filters become more popular for network applications, a membership query for a dynamic set is also required. Some network applications require high-speed processing of packets. For this purpose, Bloom filters should reside in a fast and small memory, SRAM. In this case, due to the limited memory size, stale data in the Bloom filter should be deleted to make space for new data. Namely the Bloom filter needs aging like LRU caching. In this paper, we propose a new aging scheme for Bloom filters. The proposed scheme utilizes the memory space more efficiently than double buffering, the current state of the art. We prove theoretically that the proposed scheme outperforms double buffering. We also perform experiments on real Internet traces to verify the effectiveness of the proposed scheme.

Index Terms—Bloom filter, LRU, cache, packet processing.

1 Introduction

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Bloom filters allow false positives (nonmembers misclassified as members), but the space savings often outweigh this drawback when the probability of an error is controlled. Bloom filters have been used in database applications since 1970s, but in recent years, they have become popular in networking areas [1], [2].

Traditional Bloom filters are designed for static sets. It means that a set is given ahead and all the members of the set are programmed into the bit array of the Bloom filter. Then, a membership query is executed for a given value. In this sense, Bloom filters support the operations of $insert$ and $query$, but not $delete$.

Some applications in networking area require Bloom filters to work on dynamic sets as well as static ones. Being dynamic, the set has too many members to be programmed into the Bloom filter at a time. As Bloom filters are popularly used to store packet information in a high-speed networking environment, they are required to work efficiently on dynamic sets as well as static sets. One example of dynamic sets is the approximate cache, as in [3]. In such applications, stale data should be deleted from the Bloom filter so that the total number of data in the Bloom filter at any time should not exceed a certain threshold value. Otherwise, false positives happen more frequently than the allowed error ratio. Therefore, $delete$ operation should be supported to make Bloom filters work on dynamic sets.

In many applications, it is not strange that old data should be deleted in first-in-first-out (i.e., the Bloom filter ages chronologically). For this purpose, Chang et al. proposed an aging scheme, called double buffering [3], [4], where two large buffers are used alternatively. This is similar to a least recently used (LRU) cache, but data from one buffer are cleared out whenever the memory becomes full. We stress that only one buffer is activated at any time while the other is reserved for the next epoch. In this sense, double buffering is an active/standby scheme. In the following section, we will explain it in details.

In this paper, we propose active-active buffering (A^2 buffering), a new aging scheme that makes Bloom filters work on dynamic sets more efficiently than double buffering. Using the proposed scheme, Bloom filters can store more data than double buffering when the total memory size and the tolerable false positive rate are already given. The name of A^2 buffering is from the fact that it uses two buffers simultaneously, $active1$ and $active2$ buffers. The $active1$ is expected to store the recently used data. Whenever $active1$ is full, $active2$ is cleared out, and the two buffers switch their roles. We prove theoretically that A^2 buffering works better than double buffering, which is known as the current state of the art. We show that A^2 buffering can store twice as many data as double buffering in the best-case scenarios. Even in the worst-case scenarios, this can still store as many data as double buffering. Its performance is also evaluated by experiments using real Internet traffic traces.

The rest of the paper is organized as follows: Section 2 reviews the Bloom filter and discusses the related work. Section 3 presents the design of A^2 buffering scheme. Sections 4 and 5 evaluate A^2 buffering by theoretical analysis and experiments, respectively. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Bloom Filter

A Bloom filter is a data structure for representing a set in order to support membership queries [1]. The Bloom filter consists of a bit array and a set of hash functions. We denote them by $A[i](1 \le i \le m)$ and $H = \{h_i(\cdot)|1 \le i \le k, 1 \le h_i(\cdot) \le m\}$, respectively. Note that m is the total number of bits assigned for the Bloom filter and k is the total number of hash functions. Each bit of the array is initialized to zero.

We denote the set by $S = \{s_1, \ldots, s_n\}$. Every element in S should be programmed into the bit array by setting $A[h_i(s_j)]$ to one for $1 \le i \le k$ and $1 \le j \le n$. We call this $insert$. After the programming finishes, we are ready to answer such a $query$ as "is x a member of S?" To answer the question, we test every $A[h_i(x)]$ for $1 \le i \le k$. If every bit is already set to one, x is considered a member of S. There is no false negatives (members taken as nonmembers) in Bloom filters, but false positives (nonmembers taken as members) happen and its ratio can be derived theoretically as follows [2]:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{kn/m})^k, \quad (1)$$

In this paper, we call f an allowed false positive ratio since its meaning is the theoretical expectation value for the false positive ratio. Actually, Mitzenmacher shows that the ratio of zero bits to the total allocated bits is extremely concentrated around its expectation [5].

Note that Bloom filters are based on hash functions, so the length of a member's identifier does not affect the memory requirement. This feature makes Bloom filters more feasible for applications using lengthy identifiers such as IPv6 in which 128 bits are used for an address identifier [6].

In this paper, we assume that memory size m is a fixed and small value. This assumption is in accordance with recent studies, where the Bloom filters should reside in SRAM [7]. We also assume that f is already fixed to a certain value.

As m and f are fixed, the goal is to maximize n. For this, the optimal values of k and n is derived as follows [2]:

$$k = \lfloor -log_2 f \rfloor, \quad (2)$$

• *The author is with the Korea Financial Telecommunications and Clearings Institute (KFTC), 717 YokSam-Dong, KangNam-Gu, Seoul 135-758, Korea. E-mail: mkyoon@kftc.or.kr.*

Manuscript received 28 Jan. 2008; revised 22 Oct. 2008; accepted 17 May 2009; published online 29 May 2009.
Recommended for acceptance by M. Garofalakis.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-01-0066.
Digital Object Identifier no. 10.1109/TKDE.2009.136.

```
DoubleBuffering(x)
1.  if x is in the active cache then
2.      if the active cache is more than 1/2 full
3.          insert x into the warm-up cache
4.      result := true
5.  else
6.      result := false
7.      insert x into the active cache
8.      if the active cache is more than 1/2 full
9.          insert x into the warm-up cache
10. if the active cache is full
11.     switch the active cache and warm-up cache
12.     flush the warm-up cache
13. return result
```

Fig. 1. Pseudocode of double buffering.

$$n = \left\lfloor \frac{m}{k} ln2 \right\rfloor. \quad (3)$$

In this paper, we say that a Bloom filter is full when the total number of inserted data is larger than n. Note that the number of one-bits in the Bloom filter equals $\frac{m}{2}$ when the Bloom filter is full.

2.2 Bloom Filters on Dynamic Sets

To support dynamic sets, Bloom filters need *delete* operation to make space for new incoming data. For this purpose, the simplest scheme would be cold cache [3]. Once the Bloom filter is full, all data are removed. Then, the next arriving data are programmed again. The problem is that no data remain in the memory whenever *delete* occurs. This causes a load spike that is not tolerable in certain applications like real-time systems. The alternative choice would be the counting Bloom filter [8]. Using counters rather than bits, the counting Bloom filter can delete a previously inserted data. However, this cannot delete stale data selectively unless the list of old data to delete is maintained in a separate space. We may consider randomly choosing some bits to turn off, as in [9]. However, this cannot delete old data selectively.

Deng and Rafiei [10] propose Stable Bloom Filter (SBF) to support the membership checking for dynamic sets. This scheme also uses multiple hash functions. As in counting Bloom filter [8], each cell is of multiple bits, which ranges from 0 to a certain maximum value. When a new element is programmed, the corresponding cells are set to the maximum value. Like a Bloom filter, the false positive ratio is related to the ratio of zero bits. To keep the false positive ratio below the threshold, some cells are selected and their values are decreased to keep the number of zero bits larger than a certain value. However, this approach has some limitations. First, if the maximum value of a cell is large, the memory space is wasted. Note that two-bit cells decrease the number of cells by half. We stress that the number of zero cells should be large enough to keep the false positive ratio below the threshold. Second, if a cell is randomly selected and decreased to zero, this may fail the membership checking of more than one element. Note that the cells are shared by multiple elements. From our experiments, we confirm that the average cache hit ratio of SBF is smaller than that of the current state-of-the-art scheme as well as our proposed scheme.

To the best of our knowledge, double buffering is the current best scheme to support the selective deletion of old data from Bloom filters in first-in-first-out [3]. In this scheme, the memory space is divided into two independent groups: active cache and warm-up cache. In this paper, we use the terms of cache and buffer interchangeably. The size of each cache is $\frac{m}{2}$ and the allowed false positive ratio of one cache should equal to f. The details of the algorithm are shown in Fig. 1. Note that the active cache stores all the recent data and the warm-up cache is always a subset of the active cache. The optimal number of hash functions and the maximum number of programmable data for each cache are as follows:

$$k_d = \lfloor -log_2 f \rfloor, \quad (4)$$

$$n_d = \left\lfloor \frac{m}{2k_d} ln2 \right\rfloor. \quad (5)$$

Let N_d be the total number of data programmed in both the active and warm-up caches. Since the warm-up cache is just a subset of the active cache, $max(N_d) = n_d$.

We observe that two buffers of double buffering may contain few data under certain input patterns. For example, consider the following input pattern:

1. A stream of input data fills up the active cache completely. This means that the active cache would have been reset if one more distinct datum had been programmed into the cache.
2. The same input data stream repeats itself until the warm-up cache equals the active cache.
3. A distinct input datum is programmed into the active cache, which has never been seen. Therefore, this will fill up both the warm-up cache and the active cache.
4. The active cache is flushed and two caches switch their roles. Note that the new active cache is already full.
5. Another distinct input datum is programmed, which has not been seen. Then, the active cache is flushed, which makes both caches empty.

If the above scenario happens, the range of N_d becomes as follows:

$$0 \leq N_d \leq n_d. \quad (6)$$

2.3 Other Related Work

Guo et al. propose multidimension dynamic Bloom filters to support concise representation and approximate membership queries of dynamic sets [11]. The basic idea is to add a new bit array when all the previous arrays are full. It starts with one bit array (i.e., one Bloom filter). Afterward, multiple bit arrays form a bit matrix of Bloom filters. Almeida et al. also propose the scalable Bloom filters [12] for the same design goal. However, none of them consider the issue of Bloom filter aging. Instead, they assume that the total memory size is not restricted, so they do not need to consider aging. Instead of aging, a new bit array should be added whenever necessary.

Zhao et al. propose the data structure of two-dimensional bit array to estimate the number of distinct destinations connected from the same source [13]. The proposed scheme can detect scanning attacks, which are prevalent on the Internet. They use the Bloom filter of three hash functions to get column indexes. The source address of a packet is the input to the Bloom filter. For the row index, they hash the concatenated string of the source/destination addresses. We take any source address as scanner if the three columns of the source have many one-bits compared with other columns.

Although the theoretical false positive ratio is derived from (1), Hao et al. show that the number of real false positives can be reduced. The basic idea is to divide the input set into multiple subsets, and the different sets of hash functions are used for each input subset. When a new input is programmed into the Bloom filter, the choice of a hash function is invoked. The choice is determined in a greedy manner to reduce the current number of one-bits in the Bloom filter [14].

An excellent survey on the network applications of Bloom filters is available for interested readers [2]. This survey paper covers the interesting features of Bloom filter and its network applications.

TABLE 1
Notations

m	total available memory size
f	allowed false positive ratio
k	optimal number of hash functions for given m and f
n	optimal number of programmable data for given m and f
k_d	optimal number of hash functions for one buffer of double buffering, which is equal to k
n_d	optimal number of programmable data for one buffer of double buffering
N_d	number of distinct data programmed into two buffers of double buffering
f_a	allowed false positive ratio for one buffer of A^2 buffering. $f_a < f$
k_a	optimal number of hash functions for one buffer of A^2 buffering
n_a	optimal number of programmable data for one buffer of A^2 buffering
N_a	number of distinct data programmed into two buffers of A^2 buffering

3 A^2 BUFFERING

3.1 Overview

We propose a new aging scheme for Bloom filters, active-active buffering (A^2 buffering). We assume that m, the total memory size, and f, the allowed false positive ratio, are already fixed. Under this assumption, the goal of the proposed scheme is to maximize the number of data programmed into the Bloom filter while the Bloom filter ages gracefully. The notations used in this paper are listed in Table 1 for quick reference.

3.2 Algorithm

The A^2 buffering uses two buffers at the same time, which are denoted by $active1$ and $active2$, respectively. The size of each buffer is $\frac{m}{2}$. The buffer $active1$ stores the recently used data. When $active1$ becomes full, $active2$ is flushed and two buffers switch their roles.

The detailed algorithm of A^2 buffering is shown in Fig. 2. For given x, the function returns true if x is either in $active1$ or $active2$; otherwise, it returns false. When x is in $active1$, nothing happens and the function returns true. Otherwise, we also check $active2$. If x is there, it is copied from $active2$ to $active1$. In this manner, new data are always kept in $active1$. Once $active1$ is full, $active2$ is flushed out and two buffers switch their roles. Note that $active2$ remains full at any time, which increases the cache hit ratio.

The A^2 buffering has some features that make it have the edge on the double buffering. First, both $active1$ and $active2$ can store distinct data, so more data can be programmed into the memory space. Note that one buffer is always a subset of the other buffer in double buffering. Second, even in the worst-case scenario, one buffer of A^2 buffering is always full while double buffering has one half-full buffer and one empty buffer.

3.3 Parameters

For a certain value of x, we need to check both $active1$ and $active2$ one by one when A^2 buffering is used. Therefore, the false positive ratio for each $active1$ and $active2$ should be smaller than f so that the final false positive ratio remains equal to f. Otherwise, the false positive ratio of A^2 buffering will be larger than f. Let f_a denote the false positive ratio of each buffer in A^2 *buffering*. Then, the total false positive ratio should be equal to f as follows:

$$f = 1 - (1 - f_a)(1 - f_a) = 2 \times f_a - f_a^2. \tag{7}$$

Solving for f_a, we get the following equation:

$$f_a = 1 - \sqrt{1 - f}. \tag{8}$$

For each $active1$ and $active2$, let k_a and n_a be the optimal number of hash functions and the maximum number of programmable members at a time, respectively. Then, we can derive k_a and n_a as follows:

$$k_a = \lfloor -log_2 f_a \rfloor = \lfloor -log_2(1 - \sqrt{1 - f}) \rfloor, \tag{9}$$

$$n_a = \left\lfloor \frac{m}{2k_a} ln2 \right\rfloor. \tag{10}$$

Let N_a be the total number of data actually programmed into the whole memory space of A^2 buffering. The value of N_a varies depending on the input pattern. In Section 4, we will show that $max(N_a) = 2 \times n_a \approx 2 \times max(N_d)$ in the best-case scenarios and $min(N_a) = n_a + 1 \approx max(N_d)$ in the worst-case scenarios. In Section 5, we also show experimental results using real Internet traffic traces.

```
A^2 Buffering(x)
1.  if x is in the active1 cache then
2.      result := true
3.  else
4.      if x is in the active2 cache then
5.          result := true
6.      else
7.          result := false
8.      insert x into active1
9.      if the active1 is full then
10.         flush active2
11.         switch active1 and active2
12.         insert x into active1
13. return result
```

Fig. 2. Pseudocode of A^2 buffering.

4 ANALYSIS

4.1 Theoretical Analysis

In this paper, we assume that m and f are already fixed. Under this assumption, we use N_d to measure the performance of A^2 buffering since a large N_d value elevates the cache hit ratio.

Based on the equations of the previous sections, we derive the following theorems, which prove that A^2 buffering outperforms double buffering:

Theorem 1. *The difference between k_a and k_d is less than three.*

Proof. From (4) and (9), we derive the difference between k_a and k_d as follows:

$$\begin{aligned} k_a - k_d &= \lfloor -log_2(1 - \sqrt{1 - f_d}) \rfloor - \lfloor -log_2 f_d \rfloor \\ &\le (-log_2(1 - \sqrt{1 - f_d} + 1) - (-log_2 f_d - 1). \\ &= 2 + log_2(1 + \sqrt{1 - f_d}) < 3 \end{aligned} \tag{11}$$

□

Theorem 2. $n_a \approx n_d$ *for small* f.

Proof. From Theorem 1, the maximum value of $k_a - k_d$ becomes 2 (i.e., $k_a = k_d + 2$) since both k_a and k_d are integers. Note that, at this point, the double buffering can store more members than A^2 buffering by the largest degree. Similarly, the minimum value of $k_a - k_d$ is zero (i.e., $k_a = k_d$). At this point, one buffer of A^2 buffering can store as many data as one buffer of double buffering. Therefore, by (3) and (5), we can bound n_a as follows:

TABLE 2
Comparison Example of n_d and N_a ($m = 8$ KB)

f	k_d	n_d	k_a	N_a
10^{-1}	3	3785	4	$2840 \sim 5680$
10^{-2}	6	1892	7	$1623 \sim 3246$
10^{-3}	9	1261	10	$1136 \sim 2272$
10^{-4}	13	873	14	$812 \sim 1624$
10^{-5}	16	709	17	$669 \sim 1338$
10^{-6}	19	597	20	$568 \sim 1136$
10^{-7}	23	493	24	$474 \sim 948$
10^{-8}	26	436	27	$421 \sim 842$
10^{-9}	29	391	30	$379 \sim 758$
10^{-10}	33	344	34	$335 \sim 670$

TABLE 3
Comparison Example of n_d and N_a ($m = 512$ KB)

f	k_d	n_d	k_a	N_a
10^{-1}	3	484544	4	$363409 \sim 726818$
10^{-2}	6	242272	7	$207663 \sim 415326$
10^{-3}	9	161514	10	$145364 \sim 290728$
10^{-4}	13	111818	14	$103832 \sim 207664$
10^{-5}	16	90852	17	$85508 \sim 171016$
10^{-6}	19	76507	20	$72682 \sim 145364$
10^{-7}	23	63201	24	$60569 \sim 121138$
10^{-8}	26	55909	27	$53839 \sim 107678$
10^{-9}	29	50125	30	$48455 \sim 96910$
10^{-10}	33	44049	34	$42754 \sim 85508$

$$\left\lfloor \frac{m}{2(k_d+2)} ln2 \right\rfloor \leq n_a \leq \left\lfloor \frac{m}{2k_d} ln2 \right\rfloor = n_d. \quad (12)$$

□

When f is small, k_d becomes large. Then, $n_a \approx n_d$ from (12). Some applications require f to be very small in order to reduce false positive errors. For example, the experiment from [3] uses $f = 10^{-9}$.

Note that two buffers are used at the same time in A^2 buffering while only one buffer is active in double buffering. Therefore, while $max(N_d) = n_d$ in double buffering, N_a varies depending on the pattern of input stream as follows:

Theorem 3. $n_a + 1 \leq N_a \leq 2 \times n_a$.

Proof. Suppose two extrema input patterns. The first case is that the input data stream hardly repeats itself until both the $active1$ and $active2$ buffers become full. Then, the input data stream starts repeating the previous pattern. In this case, $N_a = 2 \times n_a \approx 2 \times n_d$. The second case is that the input data hardly repeat themselves until $active1$ is full. Suppose that a new distinct datum is given so that two buffers switch their roles and $active1$ contains only one member. Now, suppose that the previous data stream repeats itself until $active1$ equals $active2$ except only one member. In this case, $N_a = n_a + 1 \approx n_d + 1$. Based on these two cases, we can bound N_a as $n_a + 1 \leq N_a \leq 2 \times n_a$. For small f, it is approximately equal to $n_d + 1 \leq N_a \leq 2 \times n_d$. □

The above theorems prove that A^2 buffering outperforms double buffering. However, the actual value of N_a is dependent on the pattern of input data stream.

4.2 Numerical Example

Tables 2 and 3 show two numerical examples. They show how k_d, n_d, k_a, and N_a are determined as f varies.

In the first table, m is 8 Kbytes. The value of f changes from 10^{-1} to 10^{-10}. Note that $k_a = k_d + 1$ holds for all the cases. As proved in the previous section, the minimum value of N_a approaches n_d as f decreases. The ratio of $\frac{min(N_a)}{n_d}$ becomes larger than 0.9 when f is 10^{-3}. With $f = 10^{-6}$, the ratio becomes larger than 0.95. The ratio of $\frac{max(N_a)}{n_d}$ becomes larger than 1.8 when f is 10^{-3}. With $f = 10^{-6}$, the ratio becomes larger than 1.9.

We set m to 512 Kbytes in Table 3. Both k_d and k_a show the same patterns as Table 2. This is because the number of hash functions is dependent on only f. Although n_d and N_a are quite larger than those of Table 2, we confirm that they increase linearly with m.

Although A^2 buffering can double the number of maximum programmable data, it may take more hash operations than the double buffering. When $active1$ does not contain the queried datum, additional membership checking should be done in $active2$. Therefore, A^2 buffering may not be desirable if the cost of hash operation were high. However, the benefit of cache can compensate the increased processing power of CPU in modern data structures [15]. Since A^2 buffering increases the number of data recorded in SRAM, we can expect the increase of the hit ratio in the cache. In this sense, A^2 buffering outperforms double buffering.

5 Experiments

In this section, we evaluate the proposed scheme using real Internet traffic traces. We use packet header traces from NLANR [16]. The traces are part of the traffic that was collected on June 1st, 2004 at the OC192c Packet-over-SONET link. The link connects Indianapolis (IPLS) to Kansas City (KSCY). This data set was made available through a joint project between NLANR PMA and Internet Traffic Research Group [16].

We implement four aging schemes: A^2 buffering, double buffering [3], cold cache [3], and SBF [10]. Each scheme is applied to the same traffic trace and the source IP address is cached. For fair comparison, all the schemes are given the same amount of memory and the same allowed false positive ratio. The default values for m and f are 4 KB and 10^{-6}, respectively. For hash functions, we use MD5 [17] with different seed values.

Note that SBF may have different results as the related parameters are differently chosen: maximum cell value and the number of hash functions. We perform extensive experiments on SBF to obtain its best result. For each experiment, we use a different maximum cell value ranging from 1 to 15, and the number of hash functions from 1 to 30, independently. Finally, we pick up the best result for SBF, which is compared with A^2 buffering, double buffering, and cold cache.

We compare the performance of four schemes in terms of two factors. First, we evaluate the average hit ratio of cache, which is defined as the ratio of the number of packets whose source IP addresses can be seen in the cache to the number of packets whose source IP address has already appeared in any previous packets. The average hit ratio of cache indicates the average efficiency. Second, we count the number of resets (i.e., the number of flushing or switching between buffers). Note that a reset may cause a temporal load spike, which is not desirable for such applications as real-time systems.

In the first experiment, we vary the allowed false positive ratio f from 10^{-9} to 10^{-3}. Fig. 3 shows how the hit ratio varies as f increases. Note that the hit ratio increases with f since large f requires many bits to turn on. The result shows that the hit ratio of A^2 buffering is higher than the other schemes. The average hit ratio of SBF is relatively low, compared with other schemes. We can explain this as follows: If a cell is randomly selected and decreased to zero, this may fail the membership checking of more than one element. The advantage of SBF is that no reset occurs.

Fig. 4 compares the number of resets. This shows that resets occur most frequently by double buffering. Although cold cache causes less resets, every reset causes the memory to remove all the information accumulated. This causes a high load spike, which may not be tolerable in such applications as real-time systems.

In the second experiment, we vary the memory size m from 1 to 16 KB. We fix $f = 10^{-6}$ for this experiment. Fig. 5 shows the average

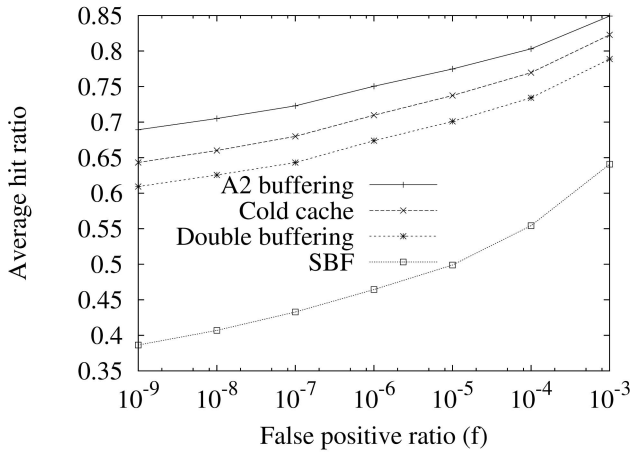

Fig. 3. Average cache hit ratio with respect to allowed false positive ratio ($m = 4$ KB).

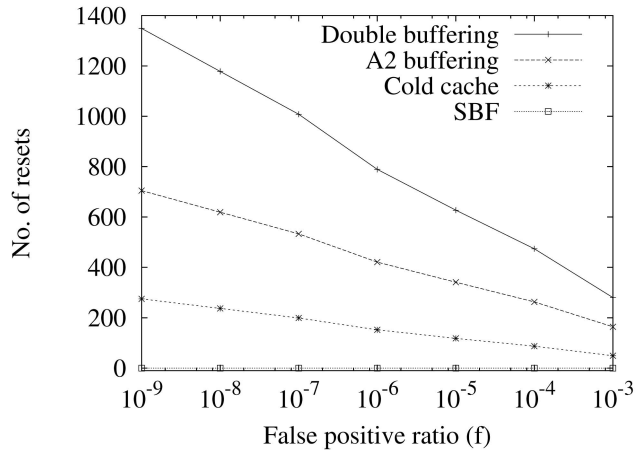

Fig. 4. Total number of resets with respect to allowed false positive ratio

hit ratio, which increases with m. Fig. 6 shows the number of resets. This graph decreases with m since large m can hold more data.

6 Conclusion

In this paper, we proposed A^2 buffering to make Bloom filters age smoothly for dynamic sets. With this scheme, we can update Bloom filters with recently used data. Consequently, Bloom filters can be useful for dynamic sets as well as static sets. We proved that A^2 buffering outperforms double buffering scheme, which is known as the best aging scheme for Bloom filters. In addition to the theoretical evaluation, we perform experiments using real Internet traffic traces and the results showed that our proposed scheme works better than any previous work.

References

[1] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, pp. 422-426, 1970.
[2] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.*, vol. 1, no. 4, pp. 485-509, June 2002.
[3] F. Chang, W. Feng, and K. Li, "Approximate Caches for Packet Classification," *Proc. IEEE INFOCOM*, Mar. 2004.
[4] W. Feng, K. Shin, D. Kandlur, and D. Saha, "The BLUE Active Queue Management Algorithms," *IEEE/ACM Trans. Networking*, vol. 10, no. 4, pp. 513-528, Aug. 2002.
[5] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking*, vol. 10, no. 5, pp. 604-612, Oct. 2002.
[6] P. Mutaf and C. Castelluccia, "Compact Neighbor Discovery: A Bandwidth Defense through Bandwidth Optimization," *Proc. IEEE INFOCOM*, Mar. 2005.

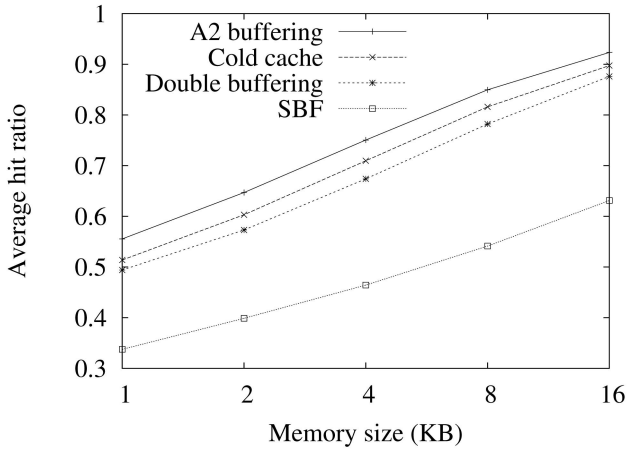

Fig. 5. Average cache hit ratio with respect to memory size ($f = 10^{-6}$).

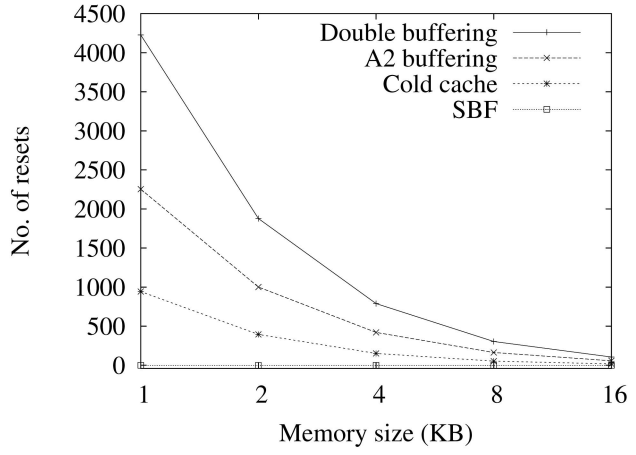

Fig. 6. Total number of resets with respect to memory Size ($f = 10^{-6}$).

[7] A. Kumar, J. Xu, L. Li, J. Wang, and O. Spatschek, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," *Proc. IEEE INFOCOM*, Mar. 2004.
[8] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking*, vol. 8, no. 3, pp. 281-293, June 2000.
[9] J. Lim and K. Shin, "Gradient-Ascending Routing via Footprints in Wireless Sensor Networks," *Proc. IEEE Int'l Real-Time Systems Symp. (RTSS '05)*, pp. 298-307, Dec. 2005.
[10] F. Deng and D. Rafiei, "Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters," *Proc. ACM SIGMOD*, June 2006.
[11] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," *Proc. IEEE INFOCOM*, Apr. 2006.
[12] P.S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, "Scalable Bloom Filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255-261, Mar. 2007.
[13] Q. Zhao, J. Xu, and A. Kumar, "Detection of Super Sources and Destinations in High-Speed Networks: Algorithms, Analysis and Evaluation," *IEEE J. Selected Areas Comm. (JSAC)*, vol. 24, no. 10, pp. 1840-1852, Oct. 2006.
[14] F. Hao, M. Kodialam, and T.V. Lakshman, "Building High Accuracy Bloom Filters Using Partitioned Hashing," *SIGMETRICS Performance Evaluation Rev.*, vol. 35, no. 1, pp. 277-288, June 2007.
[15] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures*, second ed., Silicon Press, Apr. 2007.
[16] "Passive Measurement and Analysis Project," Nat'l Laboratory for Applied Network Research (NLANR), http://nlanr.net, 2008.
[17] R. Rivest, "The MD5 Message-Digest Algorithm," IETF RFC 1321, 1992.