This paper was presented as part of the main technical program at IEEE INFOCOM 2011

One Memory Access Bloom Filters and Their Generalization

Yan Qiao Tao Li Shigang Chen
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, FL 32611, USA

***Abstract*—The Bloom filters have been extensively applied in many network functions. Their performance is judged by three criteria: processing overhead, space overhead, and false positive ratio. Due to wide applicability, any improvement to the performance of Bloom filters can potentially have broad impact in many areas of networking research. In this paper, we propose Bloom-1, a new data structure that performs membership check in one memory access, which compares favorably with the k memory accesses of a classical Bloom filter. We also generalize Bloom-1 to Bloom-g, allowing performance tradeoff between membership query overhead and false positive ratio. We thoroughly examine the variants in this new family of filters, and show that they can be configured to outperform the Bloom filters with a smaller number of memory accesses, a smaller or equal number of hash bits, and a smaller and comparable false positive ratio in practical scenarios. We also perform experiments based on a real traffic trace to support our new filter design.**

I. Introduction

The Bloom filters are compact data structures for high-speed online membership check against large data sets [1], [2]. They have wide applications in routing-table lookup [3], [4], [5], online traffic measurement [6], [7], peer-to-peer systems [8], [9], web systems [10], firewall design [11], intrusion detection [12], etc. Many network functions require membership check. A firewall may be configured with a large watch list of addresses that are collected by an intrusion detection system. If the requirement is to log all packets from those addresses, the firewall must check each arrival packet to see if the source address is a member of the list. Another example is routing-table lookup. The lengths of the prefixes in a routing table range from 8 to 32. The router can extract 25 prefixes of different lengths from the destination address of an arrival packet. It needs to determine which prefixes are in the routing tables [3], and this is a membership check problem. Some traffic measurement functions require the router to collect the flow labels [7], [13], such as source/destination address pairs or address/port tuples that identify TCP flows. Each flow label should be collected only once. When a new packet arrives, the router must check whether the flow label extracted from the packet belongs to the set that has already been collected before. In the final example for the membership check problem, we consider the CBAC (context-based access control) function in Cisco routers. When a router receives a packet, it may want to first determine whether the addresses/ports in the packet has a matching entry in the CBAC table before performing the CBAC lookup.

In all above examples, we face the same fundamental problem: For a large data set, which may be an address list, an address prefix table, a flow label set, a CBAC table, or other types of data, we want to check whether an arbitrarily given element belongs to this set or not. If there is no performance requirement, this problem can be easily solved by storing the set in a sorted array and using binary search for membership check, or keeping the set in a hash table and using linked lists to resolve hash collision. However, these approaches are inadequate if there are stringent speed and memory requirements.

Modern high-end routers and firewalls implement their per-packet operations mostly in hardware. They are able to forward each packet in a couple of clock cycles. To keep up with such high throughput, many network functions that involve per-packet processing also have to be implemented in hardware. In particular, they cannot store the data structures for membership check in DRAM because the bandwidth and delay of DRAM access cannot match the packet throughput at the line speed. Consequently, the recent research trend is to implement membership check in the high-speed on-die cache memory, which is typically SRAM. The SRAM is however small and must be shared among many online functions. This prevents us from storing a large data set directly in the form of a sorted array or a hash table. A Bloom filter is a bit array that encodes the membership of data elements in a set. Each member in the set is hashed to k bits in the array at random locations, and these bits are set to ones. To query for the membership of a given element, we also hash it to k bits in the array and see if these bits are all ones.

The performance of the Bloom filter and its many variants is judged based on three criteria: The first one is the processing overhead, which is k memory accesses and k hash operations for each membership query. The overhead limits the highest throughput that the filter can support. Because both SRAM and the hash function circuit may be shared among different network functions, it is important for them to minimize their processing overhead in order to achieve good system performance. The second performance criterion is the space overhead. Minimizing the space requirement to encode each member allows a network function to fit a large set in the limited SRAM space for membership check. The third criterion is the false positive ratio. A Bloom filter may mistakenly claim a non-member to be a member due to its lossy encoding method. There is a tradeoff between the space overhead and the false positive ratio. We can reduce the latter by allocating more

978-1-4244-9921-2/11/$26.00 ©2011 IEEE

memory.

Given the fact that the Bloom filters have been applied so extensively in the network research, any improvement to their performance can potentially have broad impact. In this paper, we design a new data structure, called *Bloom-1*, which makes just one memory access to perform membership check. Yet, it can be configured to outperform the commonly-used Bloom filter with $k = 3$. We point out that the traditional Bloom filter is not practical due to its high overhead when the optimal value of k is used to achieve a low false positive ratio. We generalize Bloom-1 to Bloom-2 and Bloom-3, which allow 2 and 3 memory accesses, respectively. We show that they can achieve the low false positive ratio of the Bloom filter with the optimal k, without incurring the same kind of high overhead. We perform thorough analysis to reveal the properties of the new family of filters proposed in this paper. We discuss how they can be applied for static or dynamic data sets. We also perform experiments based on a real traffic trace to study the performance of the new filters.

II. Bloom-1: One memory access bloom filter

A. Bloom Filter

A Bloom filter is a space-efficient data structure for membership check. It includes an array B of m bits, which are initialized to zeros. The array stores the membership information of a set as follows: Each member e of the set is mapped to k bits that are randomly selected from B through k hash functions, $H_i(e)$, $1 \leq i \leq k$, whose range is $[0, m-1)$. Kirsch and Mitzenmacher have shown that we can only use two hash functions and derive additional hash functions by a simple linear combination of the output of two hash functions [14]. To encode the membership information of e, the bits, $B[H_1(e)]$, ..., $B[H_k(e)]$, are set to ones. These are called the *membership bits* in B for the element e. Some frequently-used notations in this paper can be found in Table I.

To check the membership of an arbitrary element e', if the k bits, $B[H_i(e')]$, $1 \leq i \leq k$, are all ones, e' is considered to be a member of the set. Otherwise, it is not a member.

We can treat the k hash functions logically as a single one that produces $k \log_2 m$ hash bits. For example, suppose m is 2^{20}, $k = 3$, and a hash routine outputs 64 bits. We can extract three 20-bit segments from the first 60 bits of a single hash output and use them to locate three bits in B. Hence, from now on, instead of specifying the number of hash functions required by a filter, we will state *the number of hash bits* that are needed, which is denoted as h. The work by Kirsch and Mitzenmacher [14] gives us an efficient way to produce many hash bits, but it does not reduce the number of hash bits required by the Bloom filter.

A Bloom filter doesn't have *false negatives*, meaning that if it answers that an element is not in the set, it is truly not in the set. The filter however has *false positives*, meaning that if it answers that an element is in the set, it may not be really in the set. According to [2], the false positive ratio f_B, which is the probability of mistakenly treating a non-member as a member, is

$$f_B = (1 - (1 - \frac{1}{m})^{nk})^k \approx (1 - e^{-\frac{nk}{m}})^k, \quad (1)$$

where n is the number of members in the set. Obviously, the false positive ratio decreases as m increases, and increases as n increases. The optimal value of k (denoted as k^*) that minimizes the false positive ratio can be derived by taking the first-order derivative on (1) with respect to k, then letting the right side be zero, and solving the equation. The result is

$$k^* = \ln 2 \times m/n \approx 0.7m/n. \quad (2)$$

TABLE I
Notations

n	number of members in a set
B or $B1$	bit array
m	number of bits in the bit array B or $B1$
k	number of membership bits for each element
h	number of hash bits for locating all membership bits
k^*	the optimal value of k that minimizes the false positive ratio of a Bloom filter
$k1^*$	the optimal value of k that minimizes the false positive ratio of a Bloom-1 filter
f_B, f_{B1}, f_{Bg}	false positive ratios of a Bloom filter, a Bloom-1 filter, and a Bloom-g filter, respectively
l	number of words in a bit array
w	number of bits in a word, $m = l \times w$
$B(k = 3)$	Bloom filter that uses three bits to encode each member
$B1(k = 3)$	Bloom-1 filter that uses three bits to encode each member
$B1(h = 3 \log_2 m)$	Bloom-1 filter that uses up to $3 \log_2 m$ hash bits
B(optimal k)	Bloom filter that uses the optimal number k^* of bits to encode each member to minimize its false positive ratio
$B1$(optimal k)	Bloom-1 filter that uses the optimal number $k1^*$ of bits to encode each member to minimize its false positive ratio

B. Bloom-1 Filter

To check the membership of an element, a Bloom filter requires k memory accesses, where k is typically chosen as 3 in many papers. We introduce a new data structure, called the *Bloom-1 filter*, which requires one memory access for membership check. The basic idea is that, instead of mapping an element to k bits randomly selected from the entire bit array, we map it to k bits in a word that is randomly selected from the bit array. A word is defined as a continuous block of bits that can be fetched from the memory to the processor in one memory access. In today's computer architectures, most general-purpose processors fetch words of 32 bits or 64 bits. Specifically designed hardware may access words of 72 bits or longer.

A Bloom-1 filter is an array $B1$ of l words, each of which is w bits long. The total number m of bits is $l \times w$. To encode a member e during the filter setup, we first obtain a number of hash bits from e, and use $\log_2 l$ hash bits to map e to a word in $B1$. It is called the *membership word* of e in the Bloom-1 filter. We then use $k \log_2 w$ hash bits to further map e to k membership bits in the word and set them to ones. The total number of hash bits that are needed is $\log_2 l + k \log_2 w$. Suppose

$m = 2^{20}$, $k = 3$, $w = 2^6$, and $l = 2^{14}$. Only 32 hash bits are needed, smaller than the 60 hash bits required in the previous Bloom filter example under similar parameters.

To check if an element e' is a member in the set that is encoded in a Bloom-1 filter, we first perform hash operations on e' to obtain $\log_2 l + k \log_2 w$ hash bits. We use $\log_2 l$ bits to locate its membership word in $B1$, and then use $k \log_2 w$ bits to identify the membership bits in the word. If all membership bits are ones, it is considered to be a member. Otherwise, it is not.

The change from using k random bits in the array to using k random bits in a word may appear simple. But it is also fundamental. An important question is how it will affect the performance. A more interesting question is how it will open up new design space to configure various new filters with different performance properties. This is what we will investigate in depth.

The false negative ratio of a Bloom-1 filter is also zero. The false positive ratio f_{B1} of Bloom-1, which is the probability of mistakenly treating a non-member as a member, is derived as follows: Let F be the false positive event that a non-member e' is mistaken for a member. The element e' is hashed to a membership word. Let X be the random variable for the number of members that are mapped to the same membership word. Let x be a constant in the range of $[0, n]$, where n is the number of members in the set. Assume we use fully random hash functions. When $X = x$, the conditional probability for F to occur is

$$Prob\{F|X = x\} \quad = \quad (1 - (1 - \frac{1}{w})^{xk})^k. \tag{3}$$

Obviously, X follows the binomial distribution, $Bino(n, \frac{1}{l})$, because each of the n elements may be mapped to any of the l words with equal probabilities. Hence,

$$Prob\{X = x\} = \binom{n}{x} (\frac{1}{l})^x (1 - \frac{1}{l})^{n-x}, \quad \forall\ 0 \le x \le n. \tag{4}$$

Therefore, the false positive ratio can be written as

$$\begin{aligned} f_{B1} = Prob\{F\} &= \sum_{x=0}^{n} \left(Prob\{X = x\} \cdot Prob\{F|X = x\} \right) \\ &= \sum_{x=0}^{n} \left(\binom{n}{x} (\frac{1}{l})^x (1 - \frac{1}{l})^{n-x} (1 - (1 - \frac{1}{w})^{xk})^k \right). \end{aligned} \tag{5}$$

C. Impact of Word Size

We first investigate the impact of word size w on the performance of a Bloom-1 filter. If n, l and k are known, we can numerically obtain the optimal word size that minimizes (5). However, in reality, we can only decide the amount of memory (i.e., m) to be used for a filter, but cannot choose the word size once the hardware is installed. In the left plot of Figure 1, we compute the false positive ratios of Bloom-1 under three word sizes: 32, 64 and 72 bits, when the total amount of memory is fixed at $m = 2^{20}$. Note that the number of words, $l = \frac{m}{w}$, is inversely proportional to the word size.

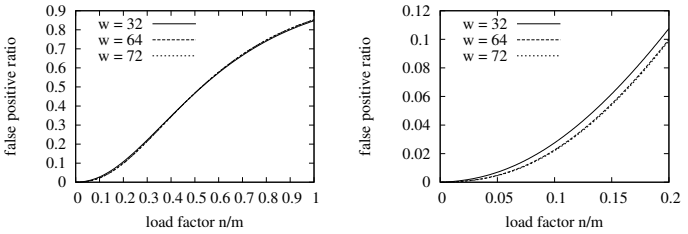

Fig. 1. *Left Plot:* False positive ratios for Bloom-1 under different word sizes. *Right Plot:* Magnified false positive ratios for Bloom-1 under different word sizes.

The horizontal axis in the figure is the *load factor*, $\frac{n}{m}$, which is the number of members stored by the filter divided by the number of bits in the filter. Since most applications require relatively small false positive ratios, we zoom in at the load-factor range of [0, 0.2] for a detailed look in the right plot of Figure 1. The result shows that comparable false positive ratios are observed for the word sizes commonly seen in existing hardware. The reason is that f_{B1} is a decreasing function in both w and l. Hence, even though a larger word size tends to decrease the false positive ratio, it also results in a smaller number of words and thus a larger number of members encoded in each word, which tends to increase the false positive ratio. These two factors somewhat cancel each other. Without losing generality, we choose $w = 64$ in our computations and simulations for the rest of the paper.

D. Bloom-1 v.s. Bloom with $k = 3$

We compare the performance of three types of filters: (1) $B(k = 3)$, which represents a Bloom filter that uses three bits to encode each member; (2) $B1(k = 3)$, which represents a Bloom-1 filter that uses three bits to encode each member; (3) $B1(h = 3 \log_2 m)$, which represents a Bloom-1 filter that uses the same number of hash bits as $B(k = 3)$ does.

Let $k1^*$ be the optimal value of k that minimizes the false positive ratio of the Bloom-1 filter in (5). It can be computed as follows: Take the first-order derivative with respect to k, then let the right side be zero, and solve the equation numerically for the optimal value of k. It may also be computed directly from (5) through bisection search or exhaustive search. Exhaustive search is not expensive because $k1^*$ is a small integer typically less than 15, as we will show shortly.

For $B1(h = 3 \log_2 m)$, we are allowed to use $3 \log_2 m$ hash bits, which can encode up to $\frac{3 \log_2 m - \log_2 l}{\log_2 w}$ membership bits in the filter, where $\log_2 l$ hash bits are used to locate the membership word and $\log_2 w$ hash bits are used to locate each membership bit in the word. However, it is not necessary to use more than the optimal number $k1^*$ of membership bits. Hence, $B1(h = 3 \log_2 m)$ actually uses $\min\{k1^*, \frac{3 \log_2 m - \log_2 l}{\log_2 w}\}$ membership bits to encode each member.

Table II compares the three filters in terms of *the number of memory accesses* and *the number of hash bits* needed for each membership query. They together represent the query overhead and control the query throughput. First, we compare $B(k = 3)$ and $B1(k = 3)$. The Bloom-1 filter saves not only memory

TABLE II
QUERY OVERHEAD COMPARISON OF BLOOM-1 FILTERS AND BLOOM FILTER WITH $k = 3$.

Data Structure	# memory access	# hash bits
$B(k=3)$	3	$3\log_2 m$
$B1(k=3)$	1	$\log_2 l + 3\log_2 w$
$B1(h=3\log_2 m)$	1	$\leq 3\log_2 m$

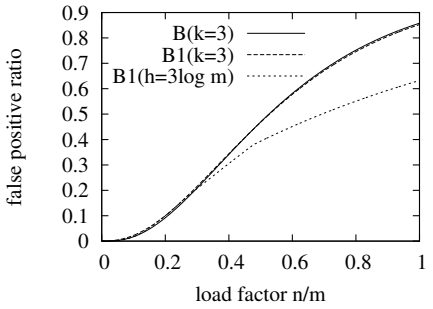

Fig. 2. Performance comparison in terms of false positive ratio. Parameters: $w = 64$ and $m = 2^{20}$.

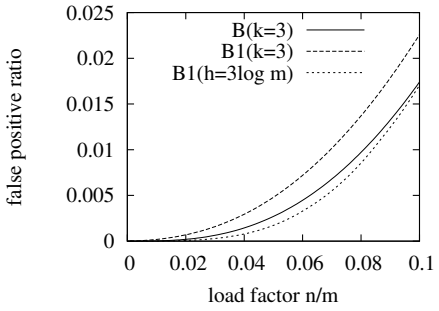

Fig. 3. Magnified false positive comparison in load-factor range of [0, 0.1].

accesses but also hash bits. For example, when $m = 2^{20}$ and $w = 64$, $B1(k = 3)$ requires about half of the hash bits needed by $B(k = 3)$. When the hash routine is implemented in hardware (such as CRC [15]), the memory access may become the performance bottleneck, particularly when the filter's bit array is located off-chip or, even if it is on-chip, the bandwidth of the cache memory is shared by other system components. In this case, the query throughput of $B1(k = 3)$ will be three times of the throughput of $B(k = 3)$.

Next, we consider $B1(h = 3\log_2 m)$. Even though it still makes one memory access to fetch a word, the processor may check more than 3 bits in the word for a membership query. If the operations of hashing, accessing memory, and checking membership bits are pipelined and the memory access is the performance bottleneck, the throughput of $B1(h = 3\log_2 m)$ will also be three times of the throughput of $B(k = 3)$.

Finally, we compare the false positive ratios of the three filters in Figure 2. We believe most real applications require small false positive ratios. Hence, we zoom in for a detailed look at the load-factor range of $[0, 0.1]$ in Figure 3, where the false positive ratios fall below 0.025. The figures show that the

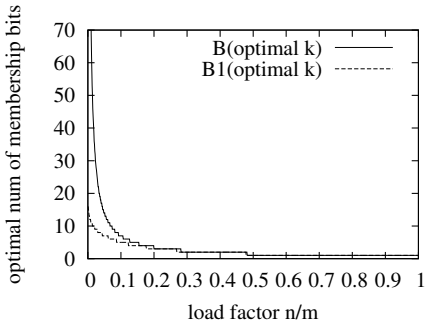

Fig. 4. Optimal number of membership bits with respect to the load factor. Parameters: $m = 2^{20}$ and $w = 64$.

TABLE III
QUERY OVERHEAD COMPARISON OF BLOOM-1 FILTER AND BLOOM FILTER WITH OPTIMAL NUMBER OF MEMBERSHIP BITS. PARAMETERS: $m = 2^{20}$ AND $w = 64$.

a. Number of memory accesses per query

	Load Factor n/m				
	0.04	0.08	0.16	0.32	0.64
B(optimal k)	17	9	4	2	1
$B1$(optimal k)	1	1	1	1	1

b. Number of hash bits per query

	Load Factor n/m				
	0.04	0.08	0.16	0.32	0.64
B(optimal k)	340	180	80	30	20
$B1$(optimal k)	62	50	38	26	20

overall performance of $B1(k = 3)$ is comparable to that of $B(k = 3)$, but its false positive ratio is slightly worse when the load factor is small. $B1(h = 3\log_2 m)$ is consistently better than $B(k = 3)$.

E. Bloom-1 v.s. Bloom with Optimal k

We can reduce the false positive ratio of a Bloom filter or a Bloom-1 filter by choosing the optimal number of membership bits. From (2), we find the optimal value k^* that minimizes the false positive ratio of a Bloom filter. From (5), we can find the optimal value $k1^*$ that minimizes the false positive ratio of a Bloom-1 filter. The values of k^* and $k1^*$ with respect to the load factor are shown in Figure 4. When the load factor is less than 0.1, $k1^*$ is significantly smaller than k^*.

We use B(optimal k) to denote a Bloom filter that uses the optimal number k^* of membership bits, and $B1$(optimal k) to denote a Bloom-1 filter that uses the optimal number $k1^*$ of membership bits. For B(optimal k), the number of memory access per membership query is k^*, and the number of hash bits needed is $k^* \log_2 m$. For $B1$(optimal k), the number of memory access per query is 1, and the number of hash bits needed is $\log_2 l + k1^* \log_2 w$.

To make the comparison more concrete, we present the numerical results of memory access overhead and hashing overhead with respect to the load factor in Table III. For example, when the load factor is 0.04, the Bloom filter requires 17 memory accesses and 340 hash bits to minimize its false positive ratio, whereas the Bloom-1 filter requires only 1 memory access and 62 hash bits. In practice, the load factor is

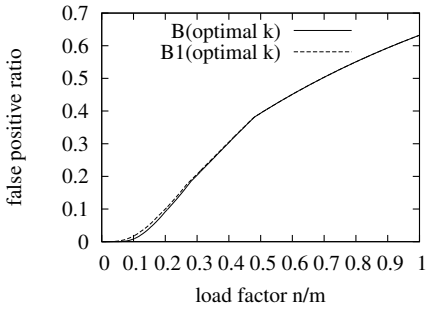

Fig. 5. False positive ratios of the Bloom filter and the Bloom-1 filter with optimal k. Parameters: $m = 2^{20}$ and $w = 64$.

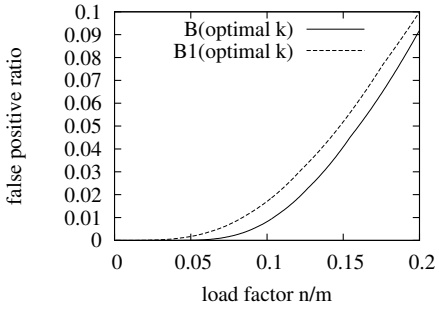

Fig. 6. Magnified false positive comparison in load-factor range of [0, 0.2].

determined by the application requirement on the false positive ratio. If an application requires a very small false positive ratio, it has to choose a small load factor.

Next, we compare the false positive ratios of $B(\text{optimal } k)$ and $B1(\text{optimal } k)$ with respect to the load factor in Figure 5. We also zoom in for a detailed look at the load factor range of [0, 0.2] in Figure 6, where the false positive ratio is below 0.1. The performance of the Bloom-1 filter is comparable to that of the Bloom filter when the load factor is not too small. But when the load factor is smaller than 0.1, the Bloom filter has a much lower false positive ratio than the Bloom-1 filter. On one hand, we must recognize the fact that, as shown in Table III, the overhead for the Bloom filter to achieve its low false positive ratio is simply too high to be practical. On the other hand, it raises a challenge for us to improve the design of the Bloom-1 filter so that it can match the performance of the Bloom filter at much lower overhead. In the next section, we generalize the Bloom-1 filter to allow performance-overhead tradeoff, which provides flexibility for practitioners to achieve a lower false positive ratio at the expense of modestly higher query overhead.

III. Bloom-g: A Generalization of Bloom-1

A. Bloom-g Filter

As a generalization of Bloom-1 filter, a *Bloom-g filter* maps each member e to g words instead of one, and spreads its k membership bits evenly in the g words. More specifically, we use $g \log_2 l$ hash bits derived from e to locate g membership words, and then use $k \log_2 w$ hash bits to locate k membership bits. The first one or multiple words are each assigned $\lceil \frac{k}{g} \rceil$ membership bits, and the remaining words are each assigned $\lfloor \frac{k}{g} \rfloor$ bits, so that the total number of membership bits is k.

When $g = k$, exactly one bit is set in each membership word. This special Bloom-k is identical to a Bloom filter with k membership bits. (Note that Bloom-k may happen to pick the same membership word more than once. Hence, just like a Bloom filter, Bloom-k allows more than one membership bit in a word.)

To check the membership of an element e', we have to access g words. Hence the query overhead includes g memory accesses and $g \log_2 l + k \log_2 w$ hash bits.

The false negative ratio of a Bloom-g filter is zero and the false positive ratio f_{Bg} of the Bloom-g filter, is derived as follows: Each member encoded in the filter randomly selects g membership words. There are n members. Together they select gn membership words (with replacement). These words are called the *encoded words*. In each encoded word, $\frac{k}{g}$ bits are randomly selected to be set as ones during the filter setup. To simplify the analysis, we use $\frac{k}{g}$ instead of taking the ceiling or floor.

Now consider an arbitrary word D in the array. Let X be the number of times this word is selected as an encoded word during the filter setup. Assume we use fully random hash functions. When any member randomly selects a word to encode its membership, the word D has a probability of $\frac{1}{l}$ to be selected. Hence, X is a random number that follows the binomial distribution $Bino(gn, \frac{1}{l})$. Let x be a constant in the range $[0, gn]$.

$$Prob\{X = x\} = \binom{gn}{x} (\frac{1}{l})^x (1 - \frac{1}{l})^{gn-x}. \tag{6}$$

Consider an arbitrary non-member e'. It is hashed to g membership words. A false positive happens when its membership bits in each of the g words are ones. Consider an arbitrary membership word of e'. Let F be the event that the $\frac{k}{g}$ membership bits of e' in this word are all ones. Suppose this word is selected for x times as an encoded word during the filter setup. We have the following conditional probability:

$$Prob\{F|X = x\} = (1 - (1 - \frac{1}{w})^{x\frac{k}{g}})^{\frac{k}{g}}. \tag{7}$$

The probability for F to happen is

$$\begin{aligned} Prob\{F\} &= \sum_{x=0}^{gn} \Big(Prob\{X = x\} \cdot Prob\{F|X = x\} \Big) \\ &= \sum_{x=0}^{gn} \left(\binom{gn}{x} \cdot (\frac{1}{l})^x \cdot (1 - \frac{1}{l})^{gn-x} \cdot (1 - (1 - \frac{1}{w})^{x\frac{k}{g}})^{\frac{k}{g}} \right) \end{aligned} \tag{8}$$

Element e' has g membership words. Hence, the false positive ratio is

$$\begin{aligned} f_{Bg} &= (Prob\{F\})^g \\ &= \left[\sum_{x=0}^{gn} \left(\binom{gn}{x} \cdot (\frac{1}{l})^x \cdot (1 - \frac{1}{l})^{gn-x} \cdot (1 - (1 - \frac{1}{w})^{x\frac{k}{g}})^{\frac{k}{g}} \right) \right]^g. \end{aligned} \tag{9}$$

TABLE IV
QUERY OVERHEAD COMPARISON OF BLOOM-2 FILTER AND BLOOM FILTER WITH $k = 3$.

Data Structure	# memory access	# hash bits
$B(k=3)$, $B3(k=3)$	3	$3\log_2 m$
$B2(k=3)$	2	$2\log_2 l + 3\log_2 w$
$B2(h=3\log_2 m)$	2	$\leq 3\log_2 m$

B. Bloom-g v.s. Bloom with $k = 3$

We compare the performance and overhead of the Bloom-g filters and the Bloom filter with $k = 3$. Because the overhead of Bloom-g increases with g, it is highly desirable to use a small value for g. Hence, we focus on Bloom-2 and Bloom-3 filters as typical examples in the Bloom-g family. We observe that increasing the value of g beyond 3 does not bring much gain in false positive ratio.

We compare the following filters: (1) $B(k = 3)$, the Bloom filter with $k = 3$; (2) $B2(k = 3)$, the Bloom-2 filter with $k = 3$; (3) $B2(h = 3\log_2 m)$, the Bloom-2 filter that is allowed to use the same number of hash bits as $B(k = 3)$ does. In this subsection, we do not consider Bloom-3 because it is equivalent to $B(k = 3)$, as we have discussed in Section III-A.

From (9), when $g = 2$, we can numerically find the optimal value of k, denoted as $k2^*$, that minimizes the false positive ratio. Similarly, when $g = 3$, we can find the optimal $k3^*$. The filter $B2(h = 3\log_2 m)$ uses $3\log_2 m$ hash bits, among which $2\log_2 l$ bits are used to locate two membership words and the remaining bits are used to locate up to $\frac{3\log_2 m - 2\log_2 l}{\log_2 w}$ membership bits. However, we shall not use more than the optimal number $k2^*$ of bits. Therefore, the number of membership bits for each element in $B2(h = 3\log_2 m)$ is set to be $\min\{k2^*, \frac{3\log_2 m - 2\log_2 l}{\log_2 w}\}$. Note that it is not expensive to exhaustively search for the value of $k2^*$ or $k3^*$ based on (9) because it must be less than the word size (e.g., 64).

Table IV compares the query overhead of three filters. The Bloom filter, $B(k = 3)$, needs 3 memory accesses and $3\log_2 m$ hash bits for each membership query. The Bloom-2 filter, $B2(k = 3)$, requires 2 memory accesses and $2\log_2 l + 3\log_2 w$ hash bits. It is easy to see $2\log_2 l + 3\log_2 w = 3\log_2 m - \log_2 l < 3\log_2 m$. Hence, $B2(k = 3)$ incurs fewer memory accesses and fewer hash bits than $B(k = 3)$. On the other hand, $B2(h = 3\log_2 m)$ uses the same number of hash bits as $B(k = 3)$ does, but makes fewer memory accesses.

Figure 7 presents the false positive ratios of $B(k = 3)$, $B2(k = 3)$ and $B2(h = 3\log_2 m)$. Figure 8 gives a detailed look at the load-factor range of [0, 0.1]. The figures show that $B(k = 3)$ and $B2(k = 3)$ have almost identical false positive ratios, whereas $B2(h = 3\log_2 m)$ performs better. For example, when the load factor is 0.04, the false positive ratio of $B(k = 3)$ is 1.5×10^{-3} and that of $B2(k = 3)$ is 1.6×10^{-3}, while the false positive ratio of $B2(h = 3\log_2 m)$ is 3.1×10^{-4}, about one fifth of the other two. Considering that $B2(h = 3\log_2 m)$ uses the same number of hash bits as $B(k = 3)$ but only 2 memory accesses per query, it is a very useful substitute of the Bloom filter to build fast and accurate data structures for membership check.

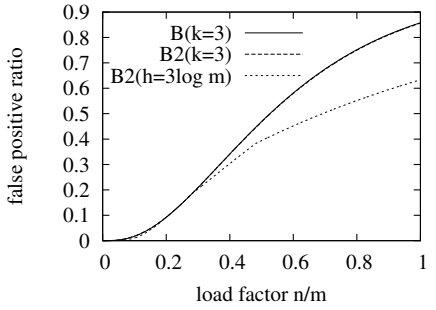

Fig. 7. False positive ratios of Bloom filter and Bloom-2 filter. Parameters: $m = 2^{20}$ and $w = 64$.

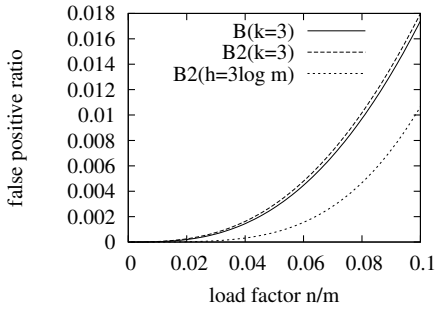

Fig. 8. Magnified false positive comparison in load-factor range of [0, 0.1].

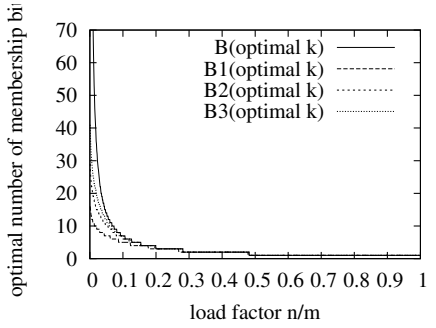

Fig. 9. Optimal number of membership bits with respect to the load factor. Parameters: $m = 2^{20}$ and $w = 64$.

C. Bloom-g v.s. Bloom with Optimal k

We now compare the Bloom-g filters and the Bloom filter when they use the optimal numbers of membership bits determined from (1) and (9), respectively. We use $B(\text{optimal } k)$ to denote a Bloom filter that uses the optimal number k^* of membership bits to minimize the false positive ratio. We use $Bg(\text{optimal } k)$ to denote a Bloom-g filter that uses the optimal number kg^* of membership bits, where $g = 1$, 2 or 3. Figure 9 compares their numbers of membership bits (i.e., k^*, $k1^*$, $k2^*$ and $k3^*$). It shows that the Bloom filter uses many more membership bits when the load factor is small.

Next we compare the filters in terms of query overhead. For $1 \leq g \leq 3$, $Bg(\text{optimal } k)$ makes g memory accesses and uses $g\log_2 l + kg^*\log_2 w$ hash bits per membership query. Numerical comparison is provided in Table V. In order to achieve a small false positive ratio, one has to keep the load factor small, which

TABLE V
QUERY OVERHEAD COMPARISON OF BLOOM FILTER AND BLOOM-g FILTER WITH OPTIMAL k. PARAMETERS: $m = 2^{20}$ AND $w = 64$.

a. Number of memory accesses per query

	Load Factor n/m				
	0.04	0.08	0.16	0.32	0.64
B(optimal k)	17	9	4	2	1
$B1$(optimal k)	1	1	1	1	1
$B2$(optimal k)	2	2	2	2	1
$B3$(optimal k)	3	3	3	2	3

b. Number of hash bits per query

	Load Factor n/m				
	0.04	0.08	0.16	0.32	0.64
B(optimal k)	340	180	80	30	20
$B1$(optimal k)	62	50	38	26	20
$B2$(optimal k)	94	70	52	40	20
$B3$(optimal k)	126	90	66	40	20

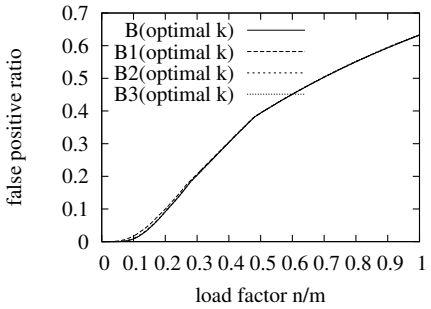

Fig. 10. False positive ratios of Bloom and Bloom-g with optimal k. Parameters: $m = 2^{20}$ and $w = 64$.

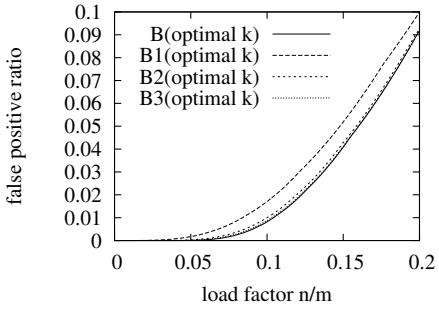

Fig. 11. Magnified false positive comparison in load-factor range of [0, 0.2].

means that B(optimal k) will have to make a large number of memory accesses and use a large number of hash bits. For example, when the load factor is 0.08, it makes 9 memory accesses with 180 hash bits per query. When the load factor is 0.04, it makes 17 memory accesses with 340 hash bits, whereas the Bloom-1, Bloom-2 and Bloom-3 filters make just 1, 2 and 3 memory accesses with 62, 94 and 126 hash bits, respectively.

Figure 10 presents the false positive ratios of the Bloom and Bloom-g filters. Figure 11 gives a magnified view for the load-factor range of $[0, 0.2]$. As we already know in Section II-E, $B1$(optimal k) performs worse than B(optimal k). But the false positive ratio of $B2$(optimal k) is very close to that of B(optimal k). Furthermore, the curve of $B3$(optimal k) is almost entirely overlapped with that of B(optimal k) for the whole load-factor range. The results indicate that we do not need to increase g beyond 3, and with just two memory accesses per query, $B2$(optimal k) works almost as good as B(optimal k), even though the latter makes many more memory accesses.

D. Discussion

The mathematical and numerical results demonstrate that Bloom-2 and Bloom-3 have smaller false positive ratios than Bloom-1 at the expense of larger query overhead. Below we give an intuitive explanation: Bloom-1 uses a single hash to map each member to a word before encoding. It is well known that a single hash cannot achieve an evenly distributed load; some words will have to encode much more members than other words, and some words may be empty as no words are mapped to them. This uneven distribution of members to the words is the reason for larger false positives. Bloom-2 maps each member to two words and splits the membership bits among the words. Bloom-3 maps each member to three words. They achieve better load balance such that most words will each encode about the same number of membership bits. This helps them improve their false positive ratios.

IV. USING BLOOM-g IN A DYNAMIC ENVIRONMENT

In order to compute the optimal number of membership bits, we must know the values of n, m, w, and l. The value of m, w and l are known once the amount of memory for the filter is allocated. The value of n is known only when the filter is used to encode a static set of members. In practice, however, the filter may be used for a dynamic set of members. For example, a router may use a Bloom filter to store a watch list of IP addresses, which are identified by the intrusion detection system as potential attackers. The router inspects the arrival packets and logs those packets whose source addresses belong to the list. If the watch list is updated once a week or at the midnight of each day, we can consider it as a static set of addresses during most of the time. However, if the system is allowed to add new addresses to the list continuously during the day, the watch list becomes a dynamic set. In this case, we do not have a fixed optimal value of k^* for the Bloom filter. One approach is to set the number of membership bits to a small constant, such as three, which limits the query overhead. In addition, we should also set the maximum load factor to bound the false positive ratio. If the actual load factor exceeds the maximum value, we allocate more memory and set up the filter again in a larger bit array.

The same thing is true for the Bloom-g filter. For a dynamic set of members, we do not have a fixed optimal number of membership bits, and the Bloom-g filter will also have to choose a fixed number of membership bits. The good news for the Bloom-g filter is that its number of membership bits is unrelated to its number of memory accesses. The flexible design allows it to use more membership bits while keeping the number of memory accesses small or even a constant one.

Comparing with the Bloom filter, we may configure a Bloom-g filter with more membership bits for a smaller false positive

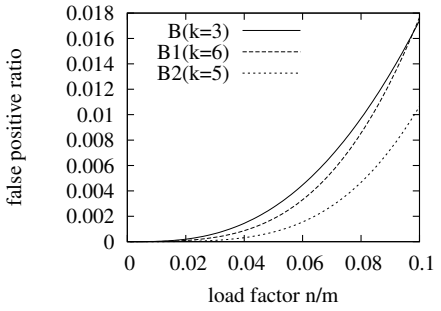

Fig. 12. False positive ratios of the Bloom filter with $k = 3$, the Bloom-1 filter with $k = 6$, and the Bloom-2 filter with $k = 5$. Parameters: $m = 2^{20}$ and $w = 64$.

ratio, while in the mean time keeping both the number of memory accesses and the number of hash bits smaller. Imagine a filter of 2^{20} used for a dynamic set of members. Suppose the maximum load factor is set to be 0.1 to ensure a small false positive ratio. Figure 12 compares the Bloom filter with $k = 3$, the Bloom-1 filter with $k = 6$, and the Bloom-2 filter with $k = 5$. As new members are added over time, the load factor increases from zero to 0.1. In this range of load factors, the Bloom-2 filter has significantly smaller false positive ratios than the Bloom filter. When the load factor is 0.04, the false positive ratio of Bloom-2 is just one fourth of the false positive ratio of Bloom. Moreover, it makes fewer memory accesses per membership query. The Bloom-2 filter uses 58 hash bits per query, and the Bloom filter uses 60 bits. The false positive ratios of the Bloom-1 filter are close to or slightly better than those of the Bloom filter. It achieves such performance by making just one memory access per query and uses 50 hash bits.

V. Experiment

We further evaluate the Bloom-g filters through experiments using real network traces.

A. Experiment Setup

Imagine that an intrusion detection system maintains a watch list consisting of previously-identified external sources, which exhibit suspicious behaviors that match the patterns of worm attacks, DDoS attacks, scanning or reconnaissance. The intrusion detection system wants to further analyze the packets from these hosts in order to capture the real offenders. It needs to match the source addresses of the incoming packets against the watch list and log the ones whose addresses are members of the list. While the whole watch list is stored in a hash table located in DRAM, it is also encoded in a Bloom or Bloom-g filter whose small size can fit in SRAM in order to keep up with the line speed. Suppose the watch list is updated once a day. It can be treated as a static list during operation.

We obtain inbound packet header traces from the main gateway at our campus. They contain 2,064,081 distinct source IP addresses and 2,192,707 distinct destination addresses. We randomly select 25,000 source addresses from the traces and place them in the watch list. We generate 10 different watch lists, perform the experiment for each of them, and average the

TABLE VI
Query overhead comparison of Bloom filter with $k = 3$ and Bloom-g filter with $h = 3\log_2 m$. Parameters: $n = 25,000$ and $w = 64$.

a. Number of memory accesses per query

data structure	# memory access
$B(k = 3)$	3
$B1(h = 3\log_2 m)$	1
$B2(h = 3\log_2 m)$	2

b. Number of hash bits per query

	m		
data structure	125Kb	250Kb	500Kb
$B(k = 3)$	51	54	57
$B1(h = 3\log_2 m)$	29	42	55
$B2(h = 3\log_2 m)$	40	54	56

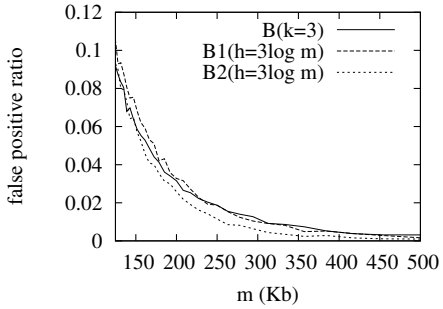

Fig. 13. False positive ratios of Bloom with $k = 3$ and Bloom-g with $h = 3\log_2 m$ in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

results. We feed the traffic traces through the filters as well as the hash table to identify the matching source addresses and compute false positive ratios based on the number of matches by the filters and the number of matches by the hash table.

Each experiment consists of two phases: the initialization phase and the execution phase. In the initialization phase, we set up the Bloom/Bloom-g filters, as well as the hash table, by using a watch list of addresses. We always allocate the same amount of memory to the Bloom filters and the Bloom-g filters for fair comparison. We use six filters. They are (a) two Bloom filters: $B(k = 3)$ and B(optimal k), and (b) four Bloom-g filters: $B1(h = 3\log_2 m)$, $B2(h = 3\log_2 m)$, $B1$(optimal k), and $B2$(optimal k). Their definitions can be found in Section II and Section III.

In the execution phase, we perform a membership query in each filter for the source address of each packet. If a filter claims that it is a member but the source is not found in the hash table, it is a false positive.

B. Performance Comparison of Bloom and Bloom-g

First, we compare the performance of $B(k = 3)$, $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$. We vary the size m of the filters from 125Kb to 500Kb, which translates into 5 to 20 bits per member in the watch list. Let $w = 64$. Table VI presents the query overhead of the Bloom filter and the Bloom-g filters. $B(k = 3)$, $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$ requires 3, 1 and 2 memory accesses respectively for each query. $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$ also require less hash bits than $B(k = 3)$. For example, when $m = 125$Kb, $B(k = 3)$

TABLE VII
QUERY OVERHEAD COMPARISON OF BLOOM FILTER AND BLOOM-g FILTER WITH OPTIMAL k. PARAMETERS: $n = 25,000$ AND $w = 64$.

a. Number of memory accesses per query

	m		
data structure	125Kb	250Kb	500Kb
B(optimal k)	3	7	14
$B1$(optimal k)	1	1	1
$B2$(optimal k)	2	2	2

b. Number of hash bits per query

	m		
data structure	125Kb	250Kb	500Kb
B(optimal k)	51	126	266
$B1$(optimal k)	29	42	55
$B2$(optimal k)	40	60	86

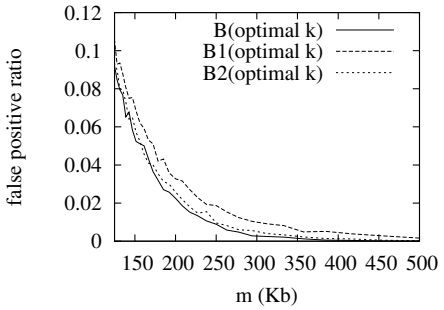

Fig. 14. False positive ratios of Bloom and Bloom-g with optimal k in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

requires 51 hash bits per query, while $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$ only need 29 and 40 hash bits respectively for each query.

Figure 13 presents the false positive ratios of the filters with respect to the amount of memory m. As our theoretical analysis has predicted, the false positive ratio of $B2(h = 3\log_2 m)$ is smaller than that of $B(k = 3)$. $B1(h = 3\log_2 m)$ also has a slightly smaller false positive ratio than $B(k = 3)$ when m is larger than 250Kb. When m is smaller than 250Kb, it yields a slightly larger false positive ratio than $B(k = 3)$. Given that the throughput of $B1(h = 3\log_2 m)$ can potentially be up to three times that of $B(k = 3)$, it is an attractive option in practice despite of its slightly higher false positive ratio.

Next, we compare B(optimal k), $B1$(optimal k) and $B2$(optimal k). Table VII presents the query overhead of the Bloom filter and Bloom-g filters. When m varies from 125Kb to 500Kb, the query overhead of B(optimal k) increases dramatically. When $m = 500$Kb, for example, B(optimal k) requires 14 memory accesses and 266 hash bits for each query, making it impractical. In comparison, $B1$(optimal k) requires only one memory access and 55 hash bits, while $B2$(optimal k) requires just two memory accesses and 86 hash bits. They remain practical solutions under this setting.

Figure 14 presents the false positive ratios of the filters with respect to m. The false positive ratio of $B2$(optimal k) is comparable to that of B(optimal k) even though its query overhead is much smaller. It is an excellent candidate for applications that require a very low false positive ratio. The false positive ratio of $B1$(optimal k) is larger than that of $B2$(optimal k), but has even smaller overhead, which represents a performance-overhead tradeoff.

VI. CONCLUSION

In this paper, we propose one memory access Bloom filters and their generalization. The new family of data structures enriches the design space of the Bloom filters and their application scope by reducing the query overhead to allow high throughput. Using a number of random bits in a word instead of from the entire bit array, we analyze the impact of this design change in terms of overhead and performance. This change also opens the door for constructing other variants for performance tradeoff. In this enlarged design space, we can configure filters that not only make fewer memory accesses but also have comparable or superior false positive ratios in scenarios where the classical Bloom filter with the optimal value of k incurs too much overhead to be practical.

REFERENCES

[1] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: a Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[3] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest Prefix Matching Using Bloom Filters," *In Proc. of ACM SIGCOMM*, August 2003.

[4] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: an Aid to Network Processing," *In Proc. of ACM SIGCOMM*, August 2005.

[5] H. Song, F. Hao, M. Kodialam, and T. Lakshman, "IPv6 Lookups Using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," *In Proc. of IEEE INFOCOM*, April 2009.

[6] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," *In Proc. of IEEE INFOCOM*, March 2004.

[7] Y. Lu and B. Prabhakar, "Robust Counting via Counter Braids: an Error-Resilient Network Measurement Architecture," *In Proc. of IEEE INFOCOM*, April 2009.

[8] P. Reynolds and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *In Proc. of the International Middleware Conference*, June 2003.

[9] A. Kumar, J. Xu, and E. Zegura, "Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks," *In Proc. of IEEE INFOCOM*, March 2005.

[10] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[11] L. Maccari, R. Fantacci, P. Neira, and R. Gasca, "Mesh Network Firewalling with Bloom Filters," *IEEE International Conference on Communications*, June 2007.

[12] D. Suresh, Z. Guo, B. Buyukkurt, and W. Najjar, "Automatic Compilation Framework for Bloom Filter Based Intrusion Detection," *Reconfigurable Computing: Architectures and Applications*, vol. 3985, pp. 413–418, 2006.

[13] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: a Novel Counter Architecture for Per-Flow Measurement," *In Proc. of ACM SIGMETRICS*, June 2008.

[14] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," *In Proc. of the 14th conference on Annual European Symposium*, September 2006.

[15] M. K. F. Hao and T. Lakshman, "Building High Accuracy Bloom Filters Using Partitioned Hashing," *In Proc. of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, June 2007.