One-Hashing Bloom Filter

Jianyuan Lu†, Tong Yang†, Yi Wang†, Huichen Dai†, Linxiao Jin†, Haoyu Song§ and Bin Liu†
† Tsinghua National Laboratory for Information Science and Technology
† Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
§ Huawei Technologies, USA

***Abstract*—Bloom filters are widely used in many network applications but the high computation cost limits the system performance. In this paper, we introduce a new variation of Bloom filter named One-Hashing Bloom Filter (OHBF) to solve the problem. OHBF requires only one base hash function plus a few simple operations to implement a Bloom filter. While keeping nearly the same theoretical false positive ratio as an ideal Bloom filter, OHBF significantly reduces the hash computation overhead. We show that the false positive performance of a standard Bloom filter implementation strongly relies on the selection of hash functions, even if these hash functions are considered good. In contrast, OHBF presents consistently better performance with a proven mathematical foundation. OHBF is ideal for high throughput and low latency applications. As OHBF is a fundamental technique in Bloom filter theory, it can be applied to many other Bloom filter variations, such as Counting Bloom Filter and Space-Code Bloom Filter.**

I. Introduction

The recent trends of Software Defined Networking (SDN) and Network Function Virtualization (NFV) [1] increasingly demand implementing and deploying network functions in software appliance such as commodity servers for flexibility and cost efficiency. Hash Table is an indispensable and powerful tool to realize a wide range of network functions. Bloom filter, as a memory efficient hashing scheme, has found its applications throughout all layers of network stack [2]. Extensive research has been conducted to improve this classical data structure in the past few years. Various novel network applications are made possible by the clever use of Bloom filter variants [3, 4, 5].

While the memory efficiency is a given benefit, the successful use of Bloom filter does come with a cost. A Bloom filter needs to use a relatively large number of hash functions (*e.g.* 35 as in [6]) and conduct the same number of memory accesses. To achieve the theoretical performance bound of a Bloom filter, these hash functions need to be strong (*i.e.* presenting good randomness and uniformity) and mutual independent. Unfortunately, good hash functions (*e.g.* MD5 and SHA-1) are known to be computation-intensive. While the memory access latency can be hidden with well established mechanisms such as caching and banking, hash computations alone consume a lot of CPU cycles and introduce excessive latency which can become the system performance bottleneck. For example, a moderate 10GE interface requires 15Mpps throughput. This leaves less than 300 clock cycles for a state-of-the-art 4GHz CPU to finish processing a packet. This limited clock budget simply cannot afford to compute a large number of independent and strong hash functions. Therefore, it is a critical and essential system requirement to reduce the cost of hash computation while retaining the desired hashing properties. Although simple hash functions may be alternatives when implementing Bloom filters for speedup [7, 8, 9], to the best of our knowledge, no papers guarantee the worst case of false positive for using them.

In this paper, we address the hash computation cost issue by introducing a novel algorithm that requires only one strong hash function to realize a Bloom filter. To the best of our knowledge, this is by far the most efficient approach with a proven performance bound. The resulting data structure, named One-Hashing Bloom Filter (OHBF), presents excellent false positive performance which is close to an ideal Bloom filter. Actually, we found that using many good hash functions does not necessarily lead to better false positive performance. Our experiments show that OHBF can outperform many existing practical Bloom filter implementations with more than one strong hash function.

Strictly speaking, in OHBF, the hashing process still generates k hash values as if we have k independent hash functions. The difference is, all the k hash values originate from a single hash function plus some simple modulo operations. It is proven that this method is equivalent to using k independent good hash functions. Meanwhile, the computation cost is effectively reduced to $1/k$, especially when the element size is large.

While we leave the proof of the convergence of our algorithm as future work, we extensively explore the design space covering a wide range of realistic application scenarios. It shows our algorithm, although simple, can satisfy all application requirements under different design constraints, such as memory size, element set size, and target false positive ratio.

II. Related Work

A. Bloom Filter and Variations

A Standard Bloom Filter (SBF) [10] is a bit vector of length m used to represent a set S of n elements. All the bits in the Bloom filter are initialized to zero. When an element $x \in S$

This work is supported by 863 project (2013AA013502), NSFC (61373143, 61432009, 61402254), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20131019172), Tsinghua University Initiative Scientific Research Program (20121080068), CISCO Award Fund, China Postdoctoral Science Foundation (No. 2014M550734, No. 2013M540949), and Jiangsu Future Networks Innovation Institute: Prospective Research Project on Future Networks (No. BY2013095-1-03). Corresponding Author: Bin Liu (liub@tsinhua.edu.cn)

978-1-4673-7113-1/15/$31.00 © 2015 IEEE

comes, we use k different hash functions $h_i(x), 1 \leq i \leq k$ to map the element to k random integer numbers uniformly in the range $[0, m-1]$. Then the corresponding bits are set to be one. Repeat the above process for each of the elements in set S. After all the elements are hashed to the Bloom filter bits, the Bloom filter has been successfully established.

A membership query could determine whether an element y belongs to the set S or not. If all the k corresponding bits indexed by $h_i(y)$ are ones, then $y \in S$. Otherwise, $y \notin S$. But the answer to the querying process can be false positive. Suppose that $y \notin S$, but all the k hashed bits happen to be ones and the query conclusion is that $y \in S$. The false positive ratio for SBF is

$$f_{SBF} = \left(1 - (1 - 1/m)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \quad (1)$$

A Bloom filter can be optimized and enhanced in different aspects. *1) Dynamic Updates*. While SBF only supports insertions, but no deletions, different techniques are developed in [11, 12, 13, 14] to support element deletion. *2) Counting*. If an SBF answers that an element belongs to the set, we do not know the concrete frequencies of this item in this set. This is amended in [11, 15, 16, 17] at the cost of more memory or hashing computation. *3) Scalability*. An SBF only supports static membership queries. In case the set cardinality is unknown prior to the Bloom filter construction, the Bloom filter variations in [18, 19] can be used. *4) Generalization*. [20, 21] introduce false negatives to Bloom filters. A tradeoff between false positives and false negatives makes the applications more flexible. The Bloomier filter [22] generalizes the SBF to support arbitrary function queries.

Some previous work also aims to reduce the hash computation cost. Kirsh and Mitzenmacher use two base hash functions $h_1(x)$ and $h_2(x)$ to construct additional hash functions in the form of $g_i(x) = h_1(x) + ih_2(x)$ [23]. We call this scheme Less Hashing Bloom Filter (LHBF). Since this technique cannot guarantee the independence of the synthetic hash functions[1], the false positive ratio in practice could be much higher than the theoretical expectation [8]. Song *et al.* introduce a simple method to produce k hash values using $O(lg\, k)$ seed hash functions [24]. However, the paper does not analyze the correlation of the k hash values. In [25], the authors also use one hash function to implement Bloom filters for set reconciliation between two nodes. Our work is generalized for wider application scenarios. More importantly, we propose and formally analyze a hash value generation and Bloom filter construction algorithm which can guarantee the false positive ratio to be nearly the same as that of an ideal Bloom filter.

B. Practical Hash Functions for Bloom Filter

A Bloom filter needs k uniform and independent hash functions. If the hash function properties are compromised, the actual false positive ratio can be much worse than the theoretical analysis. The hash functions used for Bloom filters mainly fall in three groups:

1) Cryptographic Hash Functions. Cryptographic hash functions have good randomness assurance, so they are popular choices for implementing Bloom filters. For example, MD5 is used in Bloom filter implementations [9, 11]. The complexity of MD5 is high. The cost of MD5 is proportional to key size. It requires 6.8 CPU cycles per byte on average [26]. The cost on hashing long keys can be prohibitive for some applications.

2) Non-cryptographic Hash Functions. Several relatively simple hash functions, such as CRC32, FNV and BKDR, are often used to implement Bloom filters [7, 8, 9]. Similarly, the computation complexity of these hash functions is proportional to the element size. While these hash functions are less computation-intensive than the cryptographic hash functions, their randomness is not as good, which translates to higher Bloom filter false positive ratios.

3) Universal Hash Functions. Hash functions can be selected from a family of hash functions with a certain mathematical property [27]. The Bloom filter implementations with these hash functions can approach the ideal false positive ratio [28]. Since the universal hash functions need to be "randomly" selected from a family, the practical implementation still need the aid of traditional hash functions (*i.e.*, cryptographic and non-cryptographic hash functions). Therefore, in the latter of this paper, we do not consider universal hash functions when implementing Bloom filters.

III. Design and Theoretical Analysis

A. Two Stages of Bloom Filter Hashing

In a Bloom filter, hash functions are used to compute the filter entry index. The process is essentially a mapping from $\mathcal{U} \to \mathcal{V}$, where $\mathcal{U}$ is the space of elements and $\mathcal{V}$ is the space of the Bloom filter. This process is often conducted in two stages:

Stage I: Hash Stage, $\mathcal{U} \to \mathcal{M}$, mapping $\mathcal{U}$ to a machine word size $\mathcal{M}$ (*e.g.*, 32-bit or 64-bit), using a hash function $h(x)$. This is the common understanding of hash functions.

Stage II: Modulo Stage, $\mathcal{M} \to \mathcal{V}$, mapping $\mathcal{M}$ to target $\mathcal{V}$, by modulo $|\mathcal{V}|$ (*i.e.* $h(x)\ mod\ m$). This is needed because $h(x)$ usually covers a larger space than the Bloom filter size m.

People usually treat the hash mapping as an integral process and do not distinguish these two stages explicitly. Because in most cases, modulo stage will always modulo a same length. But we show that the separation of hash functions can be taken advantage of to significantly simplify the Bloom filter implementation. In the latter of this paper, we use the notation $h(x)$ to represent hash stage, and $h(x)\ mod\ m$ to represent modulo stage.

The structure of OHBF is a little variation of SBF. Instead of treating the entire filter memory as one bit vector as in SBF, we partition the bit vector into k parts, where k is the number of hash functions used in SBF. The parts are purposely made

[1] An example to illustrate the correlation of the simulated hash functions is: consider $g_2(x) = h_1(x) + 2h_2(x)$ and $g_4(x) = h_1(x) + 4h_2(x)$, apparently, $g_2(x)$ and $g_4(x)$ have the same parity

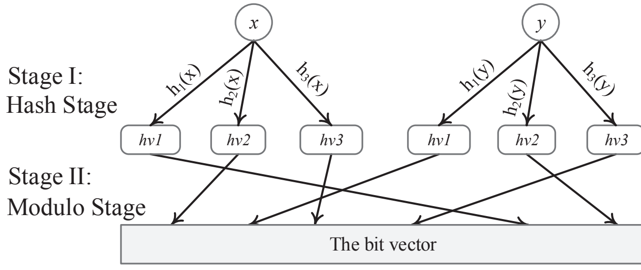


Fig. 1. A schematic view of SBF with $k = 3$ and $n = 2$. Three hash functions are used in Stage I. The output of hash functions will modulo a same length in Stage II.

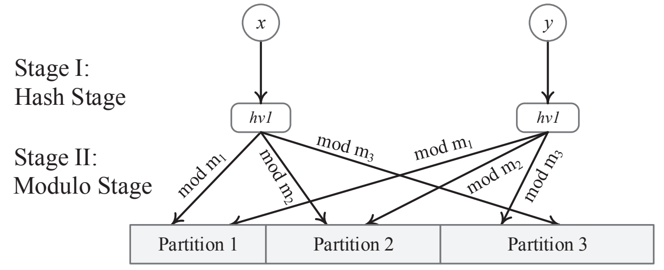


Fig. 2. A schematic view of OHBF with $k = 3$ and $n = 2$. Only one hash function is used in Stage I. The output of the one hash function will modulo different partition lengths in Stage II.

uneven. Figure 1 shows a schematic view of SBF and Figure 2 shows a schematic view of OHBF. SBF firstly uses different hash functions to get k machine words and then uses these machine words to modulo the same length m. The results can address the entire filter space. In contrast, OHBF firstly uses just one hash function to get a machine word and then uses the same machine word to modulo each partition's length. The results address one bit per partition.

We use an example to illustrate the mechanism of OHBF. Let $m_i, 1{\leq}i{\leq}k$ denotes the i_{th} partition length of OHBF. We have $m = \Sigma_{i=1}^{k} m_i$. Suppose $k = 3$, $m_1 = 11$, $m_2 = 13$, and $m_3 = 15$. When an element e comes, we apply the only hash function $h(\cdot)$ and suppose $h(e) = 4201$. As $h(e)\ mod\ m_1 = 10$, $h(e)\ mod\ m_2 = 2$, and $h(e)\ mod\ m_3 = 1$, the corresponding $10th$, $2nd$, and $1st$ bit of each partition is set.

Not any partition scheme leads to good performance. To make this scheme viable, we have to carefully choose the partition sizes for OHBF to meet several requirements:

- The partition ensures the modulo operation to generate independent values. This is proved by theoretical analysis. With this good property, we only need to find one good hash function.
- The sum of the partition sizes is close enough to the filter memory constraint. As the final size probably has a deviation from the target size, we expect the gap between them are minor enough, without noticeably affecting the target memory consumption.
- Most importantly, the resulting false positive ratio should be close enough to the equivalent SBF. The false positive ratio change due to partitions should be small enough.

All these requirements will be satisfied in our scheme. Actually, we expect the OHBF scheme is a substitute of SBF. All the parameters of OHBF should be nearly the same as SBF. Thus, we could use OHBF, anywhere needs Bloom filters, to reduce the computation cost. In the following subsections we propose a simple algorithm and provide formal mathematical analysis.

B. Proof of Independence

Let $g_i(x) = h(x)\ mod\ m_i,\ 1 \leq i \leq k$. We claim that in Stage II of OHBF hashing, $g_1(x), g_2(x) \ldots, g_k(x)$ are pairwise independent if the length of each partition satisfies:

$$(m_i, m_j) = 1,\ 1 \leq i < j \leq k \tag{2}$$

where (m_i, m_j) means the greatest common divisor of two integers m_i and m_j. Such m_i and m_j are also called relatively prime.

Before we prove our claim, we need to prove two lemmas first. To facilitate the proof, we borrow some notations from *Number Theory* and *Probability Theory*. In the following of this paper, $a|b$ denotes that a divides b; (a, b) denotes the greatest common divisor of two integers a and b; $a \equiv b\ mod\ q$ denotes a and b are congruent modulo q ; $a \not\equiv b\ mod\ q$ denotes a and b are incongruent modulo q.

Lemma 1. *If two integers $a, b \in [0, q-1]$, where $a \neq b$, $(p, q) = 1$, then $ap \not\equiv bp\ mod\ q$.*

Proof by Contradiction. Suppose that $ap \equiv bp\ mod\ q$. Let us assume $a < b$. Then $q|(bp - ap)$, which means $q|(b-a)p$. By definition $(p, q) = 1$, we can derive $q|(b - a)$. But this is impossible because $1 \leq b - a < q$. Therefore, the supposition does not hold and the statement is true. □

Lemma 2. *Let $\mathcal{Z}$ denote a uniformly distributed non-negative integer random variable over range $[0, rpq-1]$, where $r, p, q \in \mathbb{Z}^+$. Let $\mathcal{X} = (\mathcal{Z}\ mod\ p)$ and $\mathcal{Y} = (\mathcal{Z}\ mod\ q)$, where $(p, q) = 1$. Then $\mathcal{X}$, $\mathcal{Y}$ are mutually independent random variables.*

Proof. Obviously, $\mathcal{X} \in [0, p-1]$, $\mathcal{Y} \in [0, q-1]$. Let us assume $\mathcal{X} = a$, $a \in [0, p-1]$. Then, by definition, $\mathcal{Z} = cp + a$, where $c \in [0, rq-1]$. Let $\mathcal{Z}_a$ denote $\{\mathcal{Z}|\mathcal{Z} = cp+a, c \in [0, rq-1]\}$, $\mathcal{Z}_a^d$ denote $\{\mathcal{Z}|\mathcal{Z} = cp+a, c \in [dq, dq+q-1], 0 \leq d \leq r-1\}$. Then $\mathcal{Z}_a = \bigcup_{d=0}^{r-1} \mathcal{Z}_a^d$.

We first consider $c \in [0, q-1]$. By Lemma 1, we know that the q remainders $\mathcal{Z}_a^0\ mod\ q$ are not equal to each other. Note that the q remainders range in $[0, q-1]$ and they are not equal to each other, then we can say that $\mathcal{Z}_a^0\ mod\ q$ are uniformly distributed in the range $[0, q-1]$.

Then we consider $c \in [dq, dq + q - 1]$, $1 \leq d \leq r - 1$. Because $(cp + a) \equiv ((c\ mod\ q)p + a)\ mod\ q$, so the q remainders $\mathcal{Z}_a^d\ mod\ q$ are equal to $\mathcal{Z}_a^0\ mod\ q$. Therefore, $\mathcal{Z}_a^d\ mod\ q$ are also uniformly distributed in the range $[0, q-1]$.

Consequently, we can conclude that $\mathcal{Z}_a\ mod\ q$ are uniformly distributed in $[0, q-1]$. That is, $Pr(\mathcal{Y} = b|\mathcal{X} = a) =$

$Pr(\mathcal{Y} = b) = 1/q$, where $a \in [0, p-1]$, $b \in [0, q-1]$. Similarly, we can obtain that $Pr(\mathcal{X} = a|\mathcal{Y} = b) = Pr(\mathcal{X} = a) = 1/p$. Successfully, we prove that $\mathcal{X}$, $\mathcal{Y}$ are mutually independent random variables. □

Theorem 1. *Suppose that the machine word output, $\mathcal{M} = h(x)$, is uniformly distributed over $[0, rm_1m_2...m_k - 1]$. If the partition lengths $m_1, m_2, \ldots, m_k$ are pairwise relatively prime, then $g_1(x), g_2(x), \ldots, g_k(x)$ are pairwise mutually independent random variables.*

Proof. Let us assume an arbitrary pair (i, j) which satisfies that $1 \le i < j \le k$. Let $s = rm_1m_2...m_k/m_im_j$, then we know that $\mathcal{M}$ is uniformly distributed over $[0, sm_im_j - 1]$. Since $(m_i, m_j) = 1$ by definition, we get that $g_i(x)$ and $g_j(x)$ are mutually independent random variables by Lemma 2. As (i, j) is selected arbitrarily, we can conclude that the Stage II of hashing results $g_1(x), g_2(x), \ldots, g_k(x)$ are pairwise mutually independent random variables. □

In Theorem 1, we assume that $\mathcal{M}$ covers a range which is a multiple of the product $m_1m_2...m_k$. However, in practice, $\mathcal{M}$ usually covers the range a power of 2, *i.e.*, $|\mathcal{M}| = 2^L$, where L is the machine word bit width. The result that $|\mathcal{M}|$ modulo $m_1m_2...m_k$ may not be zero, which makes Theorem 1 inapplicable. There exists a simple method to solve the problem. We can discard the redundant numbers by restrict the range to $[0, c]$, where c equals to $|\mathcal{M}| - |\mathcal{M}| \% (m_1m_2 \ldots m_k)$. This implies that $|\mathcal{M}| > m_1m_2 \ldots m_k$. If $|\mathcal{M}|$ is far greater than the product $m_1m_2 \ldots m_k$, *i.e.*, $|\mathcal{M}| \gg m_1m_2 \ldots m_k$, then the modulo part can be ignored in practice.

We have proven that the Stage II hashing outputs are independent to each other, on one condition that the partition lengths are pairwise relatively prime. This result guarantees that we can eliminate the correlation of hash functions from theoretical level. With this property, we only need to find one good Stage I hash function to implement OHBF.

C. False Positive Analysis

The false positive of OHBF is caused by two factors. The first factor is the hashing collision in Stage I, denoted as event $\mathcal{E}$. If the machine words collide, it will definitely cause false positive. The second factor is the modulo collision in Stage II. If the machine words from Stage I do not collide but all the modulo remainders happen to collide, this will also cause false positive.Then the total false positive probability is:

$$\begin{aligned} f_{OHBF} = Pr(\mathcal{F}) &= Pr(\mathcal{F}|\mathcal{E})Pr(\mathcal{E}) + Pr(\mathcal{F}|\neg\mathcal{E})Pr(\neg\mathcal{E}) \\ &= Pr(\mathcal{E}) + Pr(\mathcal{F}|\neg\mathcal{E})(1 - Pr(\mathcal{E})) \end{aligned} \tag{3}$$

Suppose the machine word has L bits and L^e effective bits, where $L^e = log_2(2^L - 2^L \% m_1m_2 \ldots m_k)$ according to Theorem 1. A specific machine word will be selected with probability $\frac{1}{2^{L^e}}$, and not selected with probability $1 - \frac{1}{2^{L^e}}$. After n elements are inserted, the probability that a specific machine word has not been hashed is $\left(1 - \frac{1}{2^{L^e}}\right)^n$, which implies that the machine word collision probability is:

$$Pr(\mathcal{E}) = 1 - \left(1 - \frac{1}{2^{L^e}}\right)^n \tag{4}$$

The analysis of the second factor is similar to the machine word collision analysis. We can conclude that the false positive ratio caused by the second factor is:

$$Pr(\mathcal{F}|\neg\mathcal{E}) = \prod_{i=1}^{k} (1 - (1 - 1/m_i)^n) \tag{5}$$

Typically the machine word range (*e.g.* 2^{32} or 2^{64}) is far greater than the length of Bloom filter, *i.e.*, $2^L \gg m$. A collision of machine word is not likely happen as long as the machine word space is large enough, so $Pr(\mathcal{E})$ in practice is nearly to be 0. Therefore, the false positive probability of OHBF is simplified to be:

$$Pr(\mathcal{F}) \approx Pr(\mathcal{F}|\neg\mathcal{E}) = \prod_{i=1}^{k} (1 - (1 - 1/m_i)^n) \tag{6}$$

Because the function $\left(1 - \left(1 - \frac{1}{x}\right)^n\right)$ with respect to x is a monotonically decreasing function, we have

$$\left(1 - \left(1 - \frac{1}{\max m_i}\right)^n\right)^k \le f_{OHBF} \le \left(1 - \left(1 - \frac{1}{\min m_i}\right)^n\right)^k$$

Further, we can obtain the following theorem.

Theorem 2. *The false positive ratio of OHBF can be estimated by the following inequality,*

$$\begin{aligned} f_{OHBF} &\le \left(1 - \left(\sqrt[k]{\prod_{i=1}^{k} (1 - 1/m_i)}\right)^n\right)^k \\ &\approx \left(1 - \sqrt[k]{\prod_{i=1}^{k} e^{-\frac{n}{m_i}}}\right)^k \end{aligned} \tag{7}$$

Proof. Making use of the well-known mathematical property that the *arithmetic mean* is greater than or equal to the *geometric mean*, we can derive:

$$\begin{aligned} f_{OHBF} &= \prod_{i=1}^{k} (1 - (1 - 1/m_i)^n) \\ &\le \left(\frac{1}{k} \sum_{i=1}^{k} (1 - (1 - 1/m_i)^n)\right)^k \\ &= \left(1 - \frac{1}{k} \sum_{i=1}^{k} (1 - 1/m_i)^n\right)^k \\ &\le \left(1 - \sqrt[k]{\prod_{i=1}^{k} (1 - 1/m_i)^n}\right)^k \\ &= \left(1 - \left(\sqrt[k]{\prod_{i=1}^{k} (1 - 1/m_i)}\right)^n\right)^k \\ &\approx \left(1 - \sqrt[k]{\prod_{i=1}^{k} e^{-\frac{n}{m_i}}}\right)^k \end{aligned}$$

□

TABLE I
THEORETICAL FALSE POSITIVE PROBABILITY COMPARISON BETWEEN SBF AND OHBF

m	$n = 1000$, $k = 3$		
	SBF False Positive	OHBF False Positive	Difference (%)
10003	1.7399 e-2	1.7404 e-2	0.026
19993	2.7054 e-3	2.7058 e-3	0.014
29989	8.6273 e-4	8.6281 e-4	0.010
39995	3.7740 e-4	3.7743 e-4	0.007
49991	1.9761 e-4	1.9762 e-4	0.006
m	$n = 1000$, $k = 10$		
	SBF False Positive	OHBF False Positive	Difference (%)
10012	1.0118 e-2	1.0149 e-2	0.308
19986	8.9441 e-5	8.9612 e-5	0.192
30034	3.3187 e-6	3.3238 e-6	0.154
39994	2.8084 e-7	2.8116 e-7	0.113
49988	3.8390 e-8	3.8424 e-8	0.086

f_{OHBF} has the similar form as SBF's false positive probability $f_{SBF} = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{n}{m/k}}\right)^k$. Note that $\sqrt[k]{\prod_{i=1}^{k} e^{-\frac{n}{m_i}}}$ is the *geometric mean* of $e^{-\frac{n}{m_i}}$. This suggests that the partition size should be very close to each other. If the distribution of m_i is near m/k and m is large enough, f_{OHBF} will be very close to f_{SBF}.

Table I shows the theoretical false positive probability comparison between SBF and OHBF. It can be seen that the OHBF's false positive ratio is very close to SBF. As OHBF's partition length must meet Equation 2, the total length of OHBF may have a difference from the target filter length. If the difference is minor enough, the OHBF mechanism also works. Algorithm 1 described in the next subsection is used to determine the partitions.

D. Determine the Length of Partitions

From the previous analysis, we have known that the partition algorithm should meet the following two requirements:

- The lengths of partitions must satisfy $(m_i, m_j) = 1$, $1 \leq i < j \leq k$. Only by ensuring this, can we guarantee that the Stage II hashing in OHBF are independent to each other.
- The lengths of partitions are close to each other with small deviation. According to the false positive analysis of OHBF, this has direct impact to f_{OHBF}.

We provide a simple algorithm to satisfy these requirements: just pick k consecutive primes as the length of the partitions. We first build a prime table. The maximum prime in the table can be determined by demand. Our experience shows that in most cases it should be around $m/k + \delta$, where $\delta < 300$. Note that taking consecutive primes as the partition lengths is not necessary. We can find more simple partition method in practice.

The sums of these consecutive primes are discrete. Given a planned overall length m_p for a Bloom filter, we usually cannot get k prime numbers to make their sum m_f to be exactly m_p. As long as the difference between m_p and m_f is small enough, it neither causes any trouble for the software implementation nor noticeably shifts the false positive ratio.

Algorithm 1 Determine the Length of Partitions

Input: $m_p,\ k,\ pTable$
Output: $m_f,\ partLen$

1: scan $pTable$ to find the prime closest to $\lfloor m_p/k \rfloor$ and denote its index in $pTable$ as $pdex$
2: $sum \leftarrow 0$, $diff \leftarrow 0$, $m_f \leftarrow 0$
3: **for** $i \leftarrow pdex - k + 1$ **to** $pdex$ **do**
4: $\quad sum \leftarrow sum + pTable[i]$
5: **end for**
6: $min \leftarrow abs(sum - m_p)$
7: $j \leftarrow pdex + 1$
8: **while** true **do**
9: $\quad sum \leftarrow sum + pTable[j] - pTable[j - k]$
10: $\quad diff \leftarrow abs(sum - m_p)$
11: $\quad$ **if** $diff \geq min$ **then**
12: $\quad\quad$ **break**
13: $\quad$ **end if**
14: $\quad min \leftarrow diff$
15: $\quad j \leftarrow j + 1$
16: **end while**
17: **for** $i \leftarrow 1$ **to** k **do**
18: $\quad partLen[i] \leftarrow pTable[j - k + i]$
19: $\quad m_f \leftarrow m_f + partLen[i]$
20: **end for**

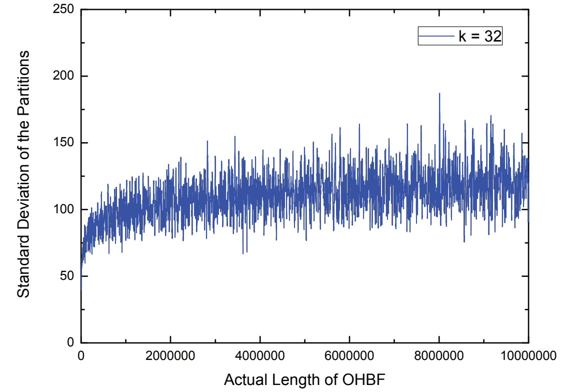


Fig. 3. Standard deviation of the partition lengths when $k = 32$

We refer to $pTable$ as our prime table, and the i_{th} number of $pTable$ is $pTable[i]$. The primes in the $pTable$ are consecutive primes in ascending order. Refer k as the number of hash functions in standard Bloom filter and refer $partLen[i]$ as the length of the i_{th} partition. We determine the length of each partition and the overall length by Algorithm 1.

Table II shows some partition examples. It can be seen that the difference between the planned length and the actual length is very small, and the length of each partition is very close. We also scan the difference between m_p and m_f when $k = 32$ and $m_p < 10,000,000$, the result shows that the biggest difference between them is only 315. The standard deviation of OHBF partition lengths when $k = 32$ is shown in Figure 3. These results illustrate that our partition algorithm can meet all the OHBF design requirements.

TABLE II
EXAMPLES OF PARTITION

m_p	m_f	Difference	Length of Each Partition for m_f ($k = 10$)									
10000	10012	1.2 e-3	971	977	983	991	997	1009	1013	1019	1021	1031
20000	19986	7.0 e-4	1973	1979	1987	1993	1997	1999	2003	2011	2017	2027
40000	39994	1.5 e-4	3947	3967	3989	4001	4003	4007	4013	4019	4021	4027
80000	80044	5.5 e-4	7949	7951	7963	7993	8009	8011	8017	8039	8053	8059
160000	159990	6.3 e-5	15937	15959	15971	15973	15991	16001	16007	16033	16057	16061
320000	319984	5.0 e-5	31957	31963	31973	31981	31991	32003	32009	32027	32029	32051
640000	640024	3.8 e-5	63929	63949	63977	63997	64007	64013	64019	64033	64037	64063
1280000	1280084	6.6 e-5	127931	127951	127973	127979	127997	128021	128033	128047	128053	128099

IV. PRACTICAL BLOOM FILTER ANALYSIS

An SBF needs k hash functions, an LHBF needs 2 hash functions, and an OHBF needs just one hash function. Through the theoretical analysis of OHBF, people may get the impression that we reduce the number of hash functions at the cost of higher false positive ratio. However, in practice, OHBF is most likely to present lower false positive ratio than SBF and LHBF. This can be explained by the fact that the practical hash functions are far worse than truly random hash functions.

All the Bloom filter analysis is based on two main assumptions on the hash functions: *1) randomness.* all the hash functions used in Bloom filters map data elements uniformly to the range, and *2) independence.* all the hash functions used in Bloom filters map data elements to the range independently. In this section, we test some representative practical hash functions on their randomness and independence. The *chi-squared test*, a well-known method in *hypothesis testing*, is introduced.

A. Chi-Squared Test

Chi-squared(χ^2) test is one of the well-known methods in *hypothesis testing* [29]. It is used to decide whether a *null hypothesis* H_0, stating that a sample frequency distribution of certain events is consistent with a particular theoretical distribution, should be rejected or not. A null hypothesis in chi-squared test is mostly like:

$$H_0 : o_1 = e_1, o_2 = e_2, \ldots, o_l = e_l \tag{8}$$

where l is the number of categories in the event, o_i is the observed sample frequency of category i, and e_i is the expected probability of category i. Usually the null hypothesis is the expectation result we want to see.

A null hypothesis is judged by a *test statistic* whether to accept or not. The test statistic of chi-squared test is:

$$S = \sum_{i=1}^{l} \frac{(n_i - ne_i)^2}{ne_i} = \sum_{all\ cells} \frac{(observed - expected)^2}{expected} \tag{9}$$

where n is the total trials and n_i is the observed trials of category i.

The *rejection region* of chi-squared test is

$$S \geq \chi^2_{\alpha,l-1} \tag{10}$$

where α is the test *significance level* (usually α is set to be 0.05) and $(l-1)$ is the *freedom degrees* of chi-squared distribution. The hypothesis H_0 would be rejected if the test statistic falls into the rejection region; otherwise, H_0 would be accepted.

B. Randomness Test of Single Hash Function

The hash functions used in Bloom filters will first map an element to a machine word of L bits. If we directly evaluate whether the hash value is uniform distribution or not, the sample space size will be 2^L. The huge size sample space will make the evaluation process very complex. Alternatively, to simplify the problem, we convert the hash value's uniform distribution test into the hash value bits sum's *binomial* distribution test.

If a hash function is truly random, then each bit of the hash value X should be *0-1* distribution, *i.e.*, $Pr(x_i = 0) = 1/2, Pr(x_i = 1) = 1/2, 1 \leq i \leq L$. And the sum of all bits, $H_X = \sum_{i=1}^{L} x_i$, should be *binomial* distribution. The probability distribution of H_X is:

$$Pr(H_X = i) = \frac{C(L,i)}{2^i}, 0 \leq i \leq L \tag{11}$$

So the null hypothesis in this hypothesis test is:

H0: *The sum of hash value bits, H_X, has binomial distribution, $Pr(H_X = i) = \frac{C(L,i)}{2^i}$*

The test statistic S_X in randomness test is:

$$S_X = \sum_{i=0}^{L} \frac{\left(O(h_X^i) - E(h_X^i)\right)^2}{E(h_X^i)} \tag{12}$$

where $O(h_X^i)$ is the observed frequency when $H_X = i$, $E(h_X^i)$ is the expected frequency when $H_X = i$.

As the binomial distribution has $L+1$ values, the degrees of freedom of χ^2 distribution are L. Hence, the rejection region in randomness test is:

$$S_X \geq \chi^2_{\alpha,L} \tag{13}$$

C. Independence Test between Different Hash Functions

Now we test the independence between two hash functions. We call every two hash functions a hash function *pair*.

Similar to the last subsection, we convert the independence test into binomial goodness-of-fit test. If a pair of hash functions are independent, then the corresponding hash values

X and Y would be independent. Therefore, the *exclusive-or* operation result, $Z = X \oplus Y$, would be a uniformly distributed variable. So each bit of Z would be 0-1 distribution with $Pr(z_i = 0) = 1/2, Pr(z_i = 1) = 1/2$, where $z_i = x_i \oplus y_i, 1 \leq i \leq L$. The sum of each bit of Z, $H_Z = \sum_{i=1}^{L} z_i$, would follow binomial distribution.

$$Pr(H_Z = i) = \frac{C(L,i)}{2^i}, 0 \leq i \leq L \tag{14}$$

So the null hypothesis in this independence test is:

H0: *The sum of all bits of Z, H_Z, has binomial distribution,* $Pr(H_Z = i) = \frac{C(L,i)}{2^i}$

The test statistic S_Z in independence test is:

$$S_Z = \sum_{i=0}^{L} \frac{\left(O(h_Z^i) - E(h_Z^i)\right)^2}{E(h_Z^i)} \tag{15}$$

And, the rejection region in independence test is:

$$S_Z \geq \chi^2_{\alpha,L} \tag{16}$$

D. Hash Function Collection and Test

We collect a total of 20 hash functions, consisting of 18 non-cryptographic hash functions and 2 cryptographic hash functions (MD5 and SHA-1). The hash functions are shown in Table III. The name and source of these hash functions mainly refer to [30]. All the hash values are 32 bits, *i.e.*, $L = 32$. As the standard MD5 and SHA-1 hash values are 128 and 160 bits respectively, we convert them to 32 bits by *exclusive-or* the hash values every 32 bits. The significance level is set to be $\alpha = 0.05$. So the rejection region in both randomness test and independence test is $S \geq \chi^2_{0.05,32} = 46.194$.

TABLE III
COLLECTED HASH FUNCTIONS AND THEIR NUMBER IN THIS PAPER

APHash h1	BKDR h2	BOB h3	CRC32 h4	DEKHash h5
DJBHash h6	FNV32 h7	Hsieh h8	JSHash h9	OCaml h10
OAAT h11	PJWHash h12	RSHash h13	SBOX h14	SDBM h15
Simple h16	SML h17	STL h18	MD5 h19	SHA-1 h20

We use the internet packet trace, obtained from CAIDA [31], to evaluate the hash functions. The trace is extracted from an OC-192 link and lasts 60 minutes. It contains 2G packets, 5M different destination IP addresses, and 50M flows. Because the input key length can affect the evaluation of hash functions, we use two kinds of keys to evaluate these hash functions. The first kind of keys is the 4-byte destination IP address and the second kind of keys is the 13-byte 5-tuple IP header.

Table IV is the randomness test result of each single hash function. The notation '$+$' represents that we accept the assumption and the notation '$-$' represents that we reject the assumption. We can see that most of the non-cryptographic hash functions cannot pass through the chi-squared test, and some hash functions, such as BKDR, FNV32 and OAAT, perform better randomness property when the input keys are longer. The two cryptographic hash functions behave good randomness property, just in accordance with our instinct.

Table V is the independence test result of hash function pairs. The notation '$+$' and '$-$' represent that we accept and reject the assumption respectively. The notation $'\circ'$ means the test does not apply. It can be concluded that many hash function pairs demonstrate some correlation relationship, especially for two hash functions both with poor randomness property. The correlation could result in large deviation from desired false positive probability. And the correlation makes the choice for hash function combinations difficult.

E. Discussion

The hash functions used by Bloom filters are not truly random in practice. First, many hash functions cannot be regarded as uniformly distributed. Second, many hash function pairs have some degree of correlation. Therefore, hash function selection when implementing a Bloom filter is difficult, especially for large k. A poor selection may lead to big false positive deviation from the theoretical value, just as the statement in [28]. Although the same good hash function with different initial seeds mostly show good independence property, the use of hash functions in this way still incurs the multiplied computation cost. On the contrary, OHBF requires only one good hash function to implement a Bloom filter. The fewer practical hash functions we need, the easier we can make the right selection. Moreover, the actual false positive rate would also be closer to the theoretical value since the possible correlation of hash functions is eliminated in OHBF.

V. EVALUATION

We evaluate the OHBF scheme from the following three aspects: *1)* the cost of modulo operation, *2)* practical false positive ratio and *3)* query running time.

We compare the different Bloom filter implementations on a commodity Server with Intel Xeon CPU E5645×2 (6 cores×2 threads, 2.4GHz) and 48GB DDR3(1,333MHz, ECC) memory. The Server runs OS Linux 2.6.43 kernel(x86_64). we use the C++ Programming Language to implement the programs. The experiments use the same real-world trace as that used in section IV-D.

A. The Cost of Modulo Operation

The cost of the modulo operation cannot be ignored. However, it only applies on the fixed-size output of the Stage I hash function. We test the modulo function on our Server and find each operation needs 7.3 clock cycles on average. In contrast, the cost of the Stage I hash functions is directly proportional to the element size. For example, CRC32, MD5, SHA-1 costs 6.9, 6.8, 11.4 clock cycles per byte respectively [26]. The cost comparison between fixed-size input modulo operation and variable-length input hash functions is shown in Figure 4. Compared to common hash functions, one modulo operation is fast enough. Intuitively, OHBF will have better performance

TABLE IV
RANDOMNESS TEST OF SINGLE HASH FUNCTIONS

Function	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10	h11	h12	h13	h14	h15	h16	h17	h18	h19	h20
key_len = 4	−	−	+	+	−	−	−	−	−	−	−	−	−	+	−	−	−	−	+	+
key_len = 13	−	+	+	+	−	−	+	−	−	−	+	−	+	+	−	−	+	−	−	+

TABLE V
INDEPENDENCE TEST BETWEEN DIFFERENT HASH FUNCTIONS, THE UPPER TRIANGULAR MATRIX CORRESPONDING TO KEY_LEN=4, THE LOWER TRIANGULAR MATRIX CORRESPONDING TO KEY_LEN=13

Function	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10	h11	h12	h13	h14	h15	h16	h17	h18	h19	h20
h1	∘	−	+	+	−	−	−	+	−	−	+	−	−	+	−	−	−	−	+	+
h2	+	∘	+	+	−	−	−	−	−	−	−	−	−	+	−	−	−	−	+	+
h3	+	+	∘	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
h4	+	+	+	∘	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
h5	+	+	+	+	∘	−	−	+	−	−	+	−	−	+	−	−	−	−	+	+
h6	+	−	+	+	+	∘	−	−	−	−	+	−	−	+	−	−	−	−	+	+
h7	+	−	+	+	+	−	∘	+	−	−	−	−	−	+	−	−	−	−	+	+
h8	+	+	+	+	+	+	+	∘	−	−	+	−	+	+	+	−	−	−	+	+
h9	+	+	+	+	+	+	+	+	∘	−	+	−	−	+	−	−	−	−	+	+
h10	+	−	+	+	−	−	−	+	+	∘	−	−	−	+	−	−	−	−	+	+
h11	+	+	+	+	+	+	+	+	−	+	∘	−	−	+	−	−	+	−	+	+
h12	+	+	−	+	−	+	+	−	+	+	+	∘	−	+	−	−	−	−	+	+
h13	+	−	+	+	+	−	−	+	+	−	+	+	∘	+	−	−	−	−	+	+
h14	+	+	+	+	+	+	+	+	+	+	+	+	+	∘	+	+	+	+	+	+
h15	−	−	+	−	+	−	−	+	+	−	+	+	−	+	∘	−	−	−	+	+
h16	+	−	+	−	+	−	−	+	+	−	+	+	−	+	−	∘	−	−	+	+
h17	+	−	+	−	+	−	−	+	+	−	−	+	−	+	−	−	∘	−	+	−
h18	+	−	+	+	+	−	−	+	+	−	+	+	−	+	−	−	−	∘	+	+
h19	+	+	+	+	+	+	+	+	+	−	+	−	+	+	−	+	+	+	∘	+
h20	+	+	−	+	+	+	+	−	+	+	+	+	+	+	+	+	+	+	−	∘

gain over SBF and LHBF as the element size and the number of hash functions increase. This is confirmed by our experiments in the following subsections.

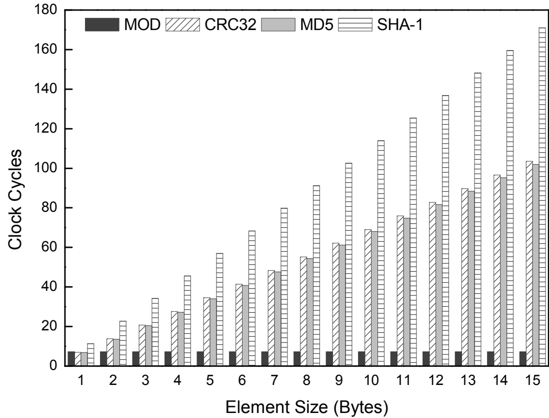


Fig. 4. The cost of fixed-size input modulo operation compared to variable-length input hash functions

B. False Positive Evaluation

We compare three Bloom filter implementations: SBF, LHBF, and OHBF. In the experiments, we set $n = 1000$. All the hash functions are selected from Table III, and their outputs are 64 bits. Other parameters will be shown in the figures. We do not run the experiments on SBF but use its ideal case as benchmark for comparison. LHBF employs the extended double hashing cube scheme discussed in [23]. The two hash functions for LHBF are MD5 and SHA-1. The one hash function for OHBF is MD5. We implement the experiments on both destination IPs (key_len=4) and 5-tuple flow identifiers (key_len=13).

The results are shown in Table VI to IX. From the four tables, we can see that both the theoretical and practical false positive rates of OHBF are very close to SBF. Although LHBF is claimed to have the same asymptotic false positive ratio as SBF when m/n is constant and $n \rightarrow \infty$, in practice, the false positive ratio difference between LHBF and SBF can be significant when n is not very large. Our experiments show that the practical false positive ratio of LHBF is much higher than SBF's theoretical false positive ratio when n is small and especially when the ideal false positive probability is very low. The reason for its poor false positive ratio is that the synthetic hash functions for LHBF have some degree of correlation, .

According to the analysis of hashing independence in section III-B, we know that the hash function at least needs 38, 40, 109, 115 hash bits for experiments corresponding to Table VI to IX respectively for OHBF. However, the hash outputs are all set to be 64 bits, which means that experiments corresponding to Table VIII to IX do not satisfy the independence requirement strictly for OHBF. However, the results of Table VIII to IX tell us that OHBF has good scalability for large number of hash functions.

C. Running Time Evaluation

In this subsection, we evaluate the computation overhead for hash functions when implementing Bloom filters. To give

TABLE VI
PRACTICAL FALSE POSITIVE RATIO COMPARISON ON KEY_LEN=4, K=3

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10003	1.740 e-2	0.026	0.029	0.046
11003	1.359 e-2	0.024	0.057	0.021
11993	1.084 e-2	0.022	0.016	0.057
13003	8.747 e-3	0.021	0.001	0.014
13993	7.186 e-3	0.019	0.038	0.036
14995	5.962 e-3	0.018	0.014	0.169
16003	4.996 e-3	0.017	0.105	0.171
17011	4.227 e-3	0.017	0.012	0.138
18005	3.616 e-3	0.016	0.006	0.147
19009	3.112 e-3	0.015	0.117	0.049

TABLE VII
PRACTICAL FALSE POSITIVE RATIO COMPARISON ON KEY_LEN=13, K=3

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10003	1.740 e-2	0.026	0.089	0.119
11993	1.084 e-2	0.022	0.027	0.116
13993	7.188 e-3	0.019	0.017	0.101
16003	4.996 e-3	0.017	0.058	0.009
18005	3.616 e-3	0.016	0.035	0.005
19993	2.706 e-3	0.014	0.014	0.138
22013	2.068 e-3	0.013	0.057	0.173
24013	1.620 e-3	0.012	0.005	0.192
26009	1.293 e-3	0.011	0.092	0.205
28001	1.049 e-3	0.010	0.063	0.134

TABLE VIII
PRACTICAL FALSE POSITIVE RATIO COMPARISON ON KEY_LEN=4, K=10

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10012	1.012 e-2	0.308	0.165	0.489
11018	5.706 e-3	0.287	0.520	0.863
11978	3.379 e-3	0.294	0.267	0.678
12990	1.991 e-3	0.250	0.356	1.609
14002	1.200 e-3	0.294	0.270	0.900
14968	7.554 e-4	0.244	0.495	1.351
16000	4.701 e-4	0.227	0.211	1.715
16974	3.059 e-4	0.243	0.017	2.282
18008	1.974 e-4	0.240	0.377	2.584
18974	1.331 e-4	0.217	0.451	2.721

TABLE IX
PRACTICAL FALSE POSITIVE RATIO COMPARISON ON KEY_LEN=13,K=10

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10012	1.012 e-2	0.308	0.212	0.489
11978	3.379 e-3	0.294	0.063	0.629
14002	1.200 e-3	0.294	0.507	1.096
16000	4.701 e-4	0.227	0.127	1.627
18008	1.974 e-4	0.240	0.457	2.668
19986	8.944 e-5	0.191	0.476	3.969
21956	4.295 e-5	0.229	0.057	5.915
24010	2.104 e-5	0.171	0.054	9.452
25998	1.102 e-5	0.185	0.222	14.93
28014	5.946 e-6	0.154	0.033	23.21

a fair comparison for different schemes, all the hash functions are BOB-based with different seeds.

Figure 5 shows the query running time comparison when the length of Bloom filters varies. The search keys are composed in a way that 50% keys are in the programmed element set and 50% keys are not. We can see from the four figures that OHBF takes the least time for querying. As the key length increasing, OHBF can spend much less time than SBF and LHBF; as the hash function number increase, OHBF spends slightly more time than LHBF and vast more time than SBF. This is because when the key becomes longer or more hash function need, more time is consumed for hash function computation. As the false positive rate decreases when m is increasing, the querying time in the same Bloom filter presents a decreasing trend. This is because the non-member keys tend to terminate the search earlier when the false positive rate is lower.

Note that sometimes not all the hash functions need to be calculated when querying an element. If the current querying bit is 0, we do not need to query the following bits. Therefore, the composition of the querying elements will affect the querying time. Figure 6 shows the querying time comparison when the percentage of member elements in the specific query set varies. It can be concluded that OHBF can spend much less time as the percentage of member elements in the specific query set increases. Moreover, the longer the keys are or the larger the hash function number is, the less time OHBF spends.

VI. CONCLUSIONS

OHBF requires only one base hash function and a set of k consecutive prime numbers as modulo operands. The composite hash functions have the strength and the desired property of k strong and independent hash functions yet the overall computation complexity is dominated by the only base hash function. The Bloom filter vector is conditioned to the selected prime numbers and each hash value addresses one of the partitions accordingly. With proven false positive performance, OHBF is ideal for applications which need both low latency and high throughput. Since OHBF is a fundamental alternative for Bloom filter implementation, it is straightforward to extend OHBF to other Bloom filter variants, such as Counting Bloom filter [11] and Space-Code Bloom Filter [16].

The simple partition algorithm discussed in the paper is easy to implement and give good performance. However, the overall actual memory size would have a small discrepancy with the target memory size. This is hardly an issue for a software system but can potentially cause trouble for a hardware implementation where embedded memory is used. Therefore, it is an interesting open question whether we can find an optimal partition that satisfies the two conditions and makes the final memory sizes meet.

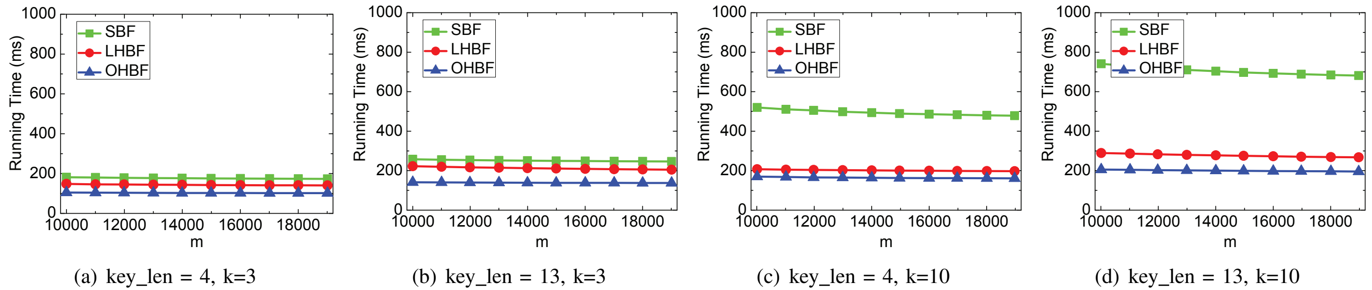
REFERENCES

[1] M. C. et.al., "Network Functions Virtualisation - Introductory White Paper," in *SDN and OpenFlow World Congress*, 2012.

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

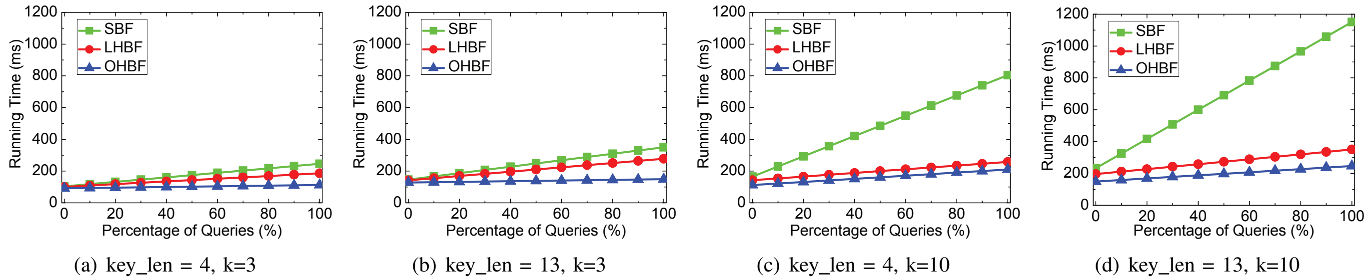
[3] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *ACM SIGCOMM*, 2003.

[4] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup using Extended Bloom Filter: An Aid to Network Processing," in *ACM SIGCOMM*, 2005.



(a) key_len = 4, k=3 (b) key_len = 13, k=3 (c) key_len = 4, k=10 (d) key_len = 13, k=10

Fig. 5. Running time of the three Bloom filters, with the length m varying. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.



(a) key_len = 4, k=3 (b) key_len = 13, k=3 (c) key_len = 4, k=10 (d) key_len = 13, k=10

Fig. 6. Running time of the three Bloom filters, with the percentage of member elements in the query set varying. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.

[5] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations," in *CoNEXT*, 2009.

[6] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *Micro, IEEE*, vol. 24, no. 1, pp. 52–61, 2004.

[7] F. Hao, M. Kodialam, and T. Lakshman, "Building High Accuracy Bloom Filters using Partitioned Hashing," in *ACM SIGMETRICS*, 2007.

[8] A. A. Iqbal, M. Ott, and A. Seneviratne, "Simplistic Hashing for Building a Better Bloom Filter on Randomized Data," in *The 13th International Conference on Network-Based Information Systems (NBiS)*, 2010.

[9] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[10] 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.

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

[12] F. Deng and D. Rafiei, "Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters," in *ACM SIGMOD*, 2006.

[13] H. Shen and Y. Zhang, "Improved Approximate Detection of Duplicates for Data Streams over Sliding Windows," *Journal of Computer Science and Technology*, vol. 23, no. 6, pp. 973–987, 2008.

[14] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. F. Magalhaes, "The Deletable Bloom Filter: A New Member of the Bloom Family," *IEEE Communications Letters*, vol. 14, no. 6, pp. 557–559, 2010.

[15] S. Cohen and Y. Matias, "Spectral Bloom Filters," in *ACM SIGMOD*, 2003.

[16] K. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-code Bloom Filter for Efficient Per-flow Traffic Measurement," in *IEEE INFOCOM*, 2004.

[17] Y. Matsumoto, H. Hazeyama, and Y. Kadobayashi, "Adaptive Bloom Filter: A Space-efficient Counting Algorithm for Unpredictable Network Traffic," *IEICE transactions on information and systems*, vol. 91, no. 5, pp. 1292–1299, 2008.

[18] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom Filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[19] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The Dynamic Bloom Filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.

[20] 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.

[21] R. P. Laufer, P. B. Velloso, D. de O Cunha, I. M. Moraes, M. D. Bicudo, M. D. Moreira, and O. Duarte, "Towards Stateless Single-packet IP Traceback," in *32nd IEEE Conference on Local Computer Networks*, 2007.

[22] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," in *The fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004.

[23] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building A Better Bloom Filter," *Random Structures & Algorithms*, vol. 33, no. 2, pp. 187–218, 2008.

[24] 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 *IEEE INFOCOM*, 2009.

[25] M. Skjegstad and T. Maseng, "Low Complexity Set Reconciliation using Bloom Filters," in *ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, 2011.

[26] "Crypto++ 5.6.0 benchmarks," 2013, http://www.cryptopp.com/benchmarks.html.

[27] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[28] M. Ramakrishna, "Practical Performance of Bloom Filters and Parallel Free-text Searching," *Communications of the ACM*, vol. 32, no. 10, pp. 1237–1239, 1989.

[29] J. L. Devore, *Probability & Statistics for Engineering and the Sciences*. Duxbury Press, 2012.

[30] C. Henke, C. Schmoll, and T. Zseby, "Empirical Evaluation of Hash Functions for Multipoint Measurements," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 39–50, 2008.

[31] C. Walsworth, E. Aben, kc claffy, and D. Andersen, "The caida anonymized 2012 internet traces," 2012, http://www.caida.org/data/passive/passive_2012_dataset.xml.