Ultra-Fast Bloom Filters using SIMD Techniques

Jianyuan Lu, Ying Wan, Yang Li, Chuwen Zhang, Huichen Dai, *Member, IEEE,* Yi Wang, *Member, IEEE,* Gong Zhang, and Bin Liu, *Senior Member, IEEE*

***Abstract*—The network link speed is growing at an ever-increasing rate, which requires all network functions on routers/switches to keep pace. Bloom filter is a widely-used membership check data structure in networking applications. Correspondingly, it also faces the urgent demand of improving the performance in membership check speed. To this end, this paper proposes a new Bloom filter variant called Ultra-Fast Bloom Filters(UFBF), by leveraging the Single Instruction Multiple Data(SIMD) techniques. We make three improvements for UFBF to accelerate the membership check speed. First, we develop a novel hash computation algorithm which can compute multiple hash functions in parallel with the use of SIMD instructions. Second, we elaborate a Bloom filter's bit-test process from sequential to parallel, enabling more bit-tests per unit time. Third, we improve the cache efficiency of membership check by encoding an element's information to a small block so that it can fit into a cache-line. We further generalize UFBF, called c-UFBF, to make UFBF supporting large number of hash functions. Both theoretical analysis and extensive evaluations show that the UFBF greatly outperforms the state-of-the-art Bloom filter variants on membership check speed.**

***Index Terms*—Bloom filter, SIMD, Parallel Techniques.**

I. Introduction

Bloom filters are kinds of space-efficient randomized data structures for membership check [1, 2]. Due to their simplicity and efficiency, Bloom filters (and their variants) have been applied in a wide range of network applications. For example, they have been used in routing table lookup [3, 4, 5], packet classification [6], network measurement [7, 8, 9], web caching [10, 11], and fast hash table lookup [12] *etc*. The trend of these applications is that they need to run in a higher and higher speed network environment. Currently, the 40GE and 100GE ports for routers' line-cards have already been commercialized and deployed [13]. High-end core routers such as Cisco CRS-X [14] and Huawei NE9000 [15] both support 400 Gbps line-card (a line-card can accommodate several high-speed ports). The development of high-speed network requires all network functions to run at line rate, leaving a very limited time budget for network devices to process every packet. For example, a 40GE port needs to achieve 60Mpps throughput, *i.e.*, processing a packet within 75 clock cycles when using a state-of-the-art 4GHz CPU. Therefore, Bloom filters need to run extremely fast to avoid becoming the network applications' performance bottleneck.

J. Lu is a joint postdoctoral research fellow in Tsinghua University and Alibaba Cloud. (e-mail: lu-jy11@mails.tsinghua.edu.cn, gaolin.ljy@alibaba-inc.com)

Y. Wan, Y. Li, C. Zhang, H. Dai, and B. Liu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. (Corresponding Author: Bin Liu. Email: liub@mail.tsinghua.edu.cn)

Y. Wang and G. Zhang are with Huawei Future Network Theory Lab, Hong Kong.

Most of the researches on network applications assume the Bloom filters have no (or, tiny) cost for membership check. However, in fact, it is not. A Bloom filter needs to compute k independent hash functions and conduct the same number of memory accesses for an element's membership check. On the one hand, the hash functions in Bloom filters have computational cost. We know that strong hash functions (*e.g.*, MD5 and SHA-1) are computation-intensive [16]. Though simple hash functions can be the alternatives for Bloom filters [17], their performance would be lower and the computational cost shall not be neglected. A test on our actual machine shows that *MurmurHash* (a simple non-cryptographic hash function) consumes 23 clock cycles on average for one hash computation. On the other hand, the memory accesses in Bloom filters have time cost and in some cases may cause several cache misses. Due to the limited on-chip cache size, most of the system data is stored on off-chip storage (*e.g.*, DRAM). In the worst case, k cache misses will occur in one element's membership check. A large amount of cache misses will deteriorate the system performance to a large extent. Systems which employ a large number of Bloom filters(*e.g.*, 24 in [3]) or use a large number of hash functions in one Bloom filter(*e.g.*, over 10 in [18]), will experience more obvious computational cost and memory access delay.

In this paper, we propose a new Bloom filter variant named Ultra-Fast Bloom Filter (UFBF), aiming to improve a Bloom filter's membership check speed. The UFBF consists of a sequence of blocks. An element's information is encoded in a randomly selected block. In practice, if the blocks are cache-line size aligned and the block size divides the cache-line size, only (at most) one cache miss would occur during an element's membership check. To speedup the hash computation in UFBF, we develop a novel algorithm which can compute the k hash functions in parallel. The algorithm uses CPU's multimedia instructions, also known as Single Instruction Multiple Data (SIMD) instructions[1] to improve the parallelism in membership check process. By setting different initial seeds in SIMD registers and implementing hash function code with SIMD instructions, we can achieve to complete computing the k hash functions in parallel. To further speedup the sequential bit-test process, we use the SIMD instructions to test k bits in parallel. To facilitate the use of SIMD instructions in bit-test process, a block in UFBF is divided into k consecutive words, and we associate each word with one hash function. That is to say, a hash function can only address its associate word.

Essentially, we make three optimizations for the UFBF to enable it runing fast. First, the UFBF reduces hash computa-

[1]The SIMD instructions are widely supported by general CPUs and embedded CPUs. Take Intel CPU as an example. It supports SIMD instruction sets, such as *MMX, SSE, AVX, FMA, KNC, SVML* [19].

tion time to (approximate) $1/k$ of a standard Bloom filter, by developing a novel hash computation algorithm which computes k hash values in parallel. Second, the UFBF changes the sequential bit-test process to parallel bit-test process. The UFBF changes the data structure to a block-word style, which brings in parallelism in membership check. Third, the UFBF improves the cache efficiency by encoding an element's information to a block. In this way, at most one cache miss would happen in an element's membership check.

The remaining of the paper is organized as follows. Section II surveys the related work. Section III details the UFBF scheme and theoretical analysis. Section IV introduces an extension of UFBF. Section V presents the experimental results for performance evaluation. Section VI concludes the paper.

II. Related Work

A. Standard Bloom Filter

Bloom filter was introduced by Burton H. Bloom in 1970 [20], which is called Standard Bloom Filter (SBF) in this paper. An SBF is a space-efficient randomized data structure which encodes a large data set to a small memory space. A lookup in SBF can only answer one question: whether an element belongs to a set or not. However, the lookup answers have false positives. A false positive happens when an SBF answers that an element belongs to a set, but actually it is not. In practical applications, the false positive probability is usually set to be very small, *e.g.*, less than 10^{-6}, to avoid a significant impact on performance. For better elaboration of SBF and our following ideas, we use a number of notations in this paper, which are shown in Table I.

An SBF for representing a set S is encoded in an array of m bits. All bits are initially set to 0. Assume $S = \{x_1, x_2, \ldots, x_n\}$ of n elements is going to be encoded in SBF. An element uses k independent hash functions $h_1, h_2, \ldots, h_k$ with range $[0, m-1]$ to select bits to set. For each element $x \in S$, the bits with location $h_i(x)$ are set to 1 for $1 \leq i \leq k$. All the bits in SBF are shared by all hash functions and all elements. So a bit in SBF may be set to 1 multiple times, but only the first set affects.

After the encoding, the main function of SBF is to implement membership queries. Given an element e, the SBF has to check whether $e \in S$ or not. If all bits with location $h_i(e)$ are set to 1, we say $e \in S$. If at least one bit with location $h_i(e)$ is 0, we say $e \notin S$. A false positive may happen in the situation that if $e \notin S$, but all bits with location $h_i(e)$ are set to 1 by elements in S.

The false positive probability can be calculated by the following formula:

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

The false positive probability can be affected by three parameters m, n, k. In practical applications, n is determined by the set size. f_s decreases as m increases, so we can lower false positive probability by increasing memory space if allowed. The most flexible parameter is k, which means how many hash functions we use in the system. We can minimize the false positive probability for fixed m and n to get an optimal hash function number k_{opt}. By taking the derivative of f_s with respect to k and equalizing it to 0, we can get:

$$k_{opt} = (m/n)ln2 \approx 9m/13n \quad (2)$$

with optimal k_{opt}, the false positive probability is:

$$f_s = \left(\frac{1}{2}\right)^{k_{opt}} \quad (3)$$

It means each bit in the bit array is set to 1 with probability $\frac{1}{2}$ when k equals to the optimal value.

TABLE I
Notations

n	number of elements in a set
m	number of bits in the bit array
k	number of membership bits for each element
w	number of bits in a word
r	number of blocks in a Bloom filter, $m = r \times w \times k$
b	number of bits in a block, $b = w \times k$
c	number of selected blocks to encode an element in c-UFBF
f_s	the false positive probability of SBF
f_u	the false positive probability of UFBF
f_c	the false positive probability of c-UFBF

B. Towards Fast Bloom Filters

Our work in this paper aims to build fast Bloom filters by using SIMD techniques. A previous work builds a vectorized implementation for probing Bloom filters using SIMD techniques [21]. However, this work is only an engineering implementation, lacking of theoretical improvements. Several previous studies attempt to build fast Bloom filters, which can be grouped into two categories:

1) Improving cache efficiency. In order to check a membership, a Bloom filter needs to perform k memory accesses. In the worst case, each memory access results in one cache miss in each element membership query. We know that high cache miss rate will deteriorate the program performance. Therefore, a few researches try to reduce cache misses in membership check to improve the Bloom filter's lookup performance. One-Memory Bloom Filter (OMBF) [22] improves the cache efficiency by restricting one element's hashing space to a word. A word is defined as the communication bandwidth between the off-chip memory and the processor in one memory access, *e.g.*, 32 bits or 64 bits. To encode an element, an OMBF first selects a word from the bit array using an additional hash function, and then maps k bits in the word using k hash functions. OMBF effectively reduces (at most) k cache misses to (at most) one cache miss in an element's membership check. However, this method increases the false positive probability compared to SBF. Blocked Bloom Filter (BBF) [23] has a similar framework with OMBF. The difference is that BBF consists of a sequence of blocks (instead of words in OMBF). BBF restricts one element's hashing space to a block, and a block has a cache-line size. A common cache-line size is 512 bits in modern CPUs. If blocks are cache-line aligned, only (at most) one cache miss could happen in one element's membership check.

2) Reducing hash computation cost. A Bloom filter needs to compute k independent hash functions to implement membership check. We know that hash computation is time-consuming. If k is large, the excessive latency introduced by hash computation will become the system performance bottleneck. Lu *et al.* [24] propose a Bloom filter variant, called One-Hashing Bloom Filter (OHBF), to lower the hash computation overhead in Bloom filters. An OHBF uses only one base hash function plus k modulo operations to implement a Bloom filter. The bit array in OHBF is divided into k partitions. To encode an element, OHBF selects a bit and sets it to 1 in each partition. The locations of selected bits are determined by using the base hash value to modulo each partition's size. Though only one base hash function is used, OHBF cannot reduce the hash computation overhead to $1/k$ as the additional modulo operations bring in excessive computation overhead. Kirsh and Mitzenmacher [25] propose Less Hashing Bloom Filter (LHBF) which uses two base hash functions $h_1(x), h_2(x)$ to implement a Bloom filter. If more than two hash functions are needed, LHBF mainly employs a form $g_i(x) = h_1(x) + i * h_2(x)$ to construct additional hash functions. The authors have proved that LHBF has the same asymptotic false positive probability as SBF. Song *et al.* introduce a simple method to produce k hash values using $O(log\, k)$ seed hash functions [4]. However, the paper lacks theoretical analysis on the randomness of the additional synthetic hash functions.

III. Ultra-Fast Bloom Filters

In this section, we introduce a new Bloom filter variant, called Ultra-Fast Bloom Filter (UFBF), which aims to improve the Bloom filter's membership check speed in practice. The UFBF improves the parallelism for hash computation and membership bit-tests by using SIMD (Single Instruction Multiple Data) instructions. Since UFBF encodes an element's information to a small block, which can be fitted into the CPU's cache-line, thus it effectively improves the cache efficiency when implementing membership check.

A. Basic Data Structure

The UFBF is composed of a sequence of r blocks, and each block has b bits. A block contains k consecutive words, and each word has w bits. Apparently, $b = k * w$. The basic structure of UFBF is shown in Figure 1. A word means a group of bits whose length equals to general registers' bit-length. For example, the length of general registers of modern CPUs is 32-bit or 64-bit, which means a word has w = 32 (or 64) bits in practice. Note that a Bloom filter has m bits in total. Therefore, we have $m = r * b = r * k * w$.

In the insertion process, an element's information is encoded in a randomly selected block. The insertion process is as follows. UFBF first selects a block from the bit array using a hash function h_0. Then it selects k bits in the selected block using k hash functions $h_1, h_2, \ldots, h_k$, and sets these bits to ones. In fact, the hash function $h_i, 1 \leq i \leq k$ is associated with word[i], and it can only address the bits in its associate word.

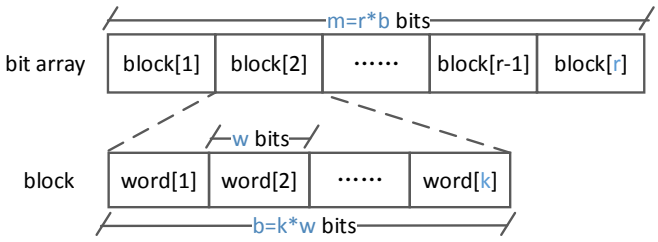


Fig. 1. The basic structure of UFBF.

We use an example to illustrate the insertion process of UFBF. Assume an element e is going to be encoded and the word size is $w = 4$. First, we select a block in the bit array, *i.e.*, block[$h_0(e)$]. Second, we select k bits from k words (in block[$h_0(e)$]) using $h_1(e) = 1, h_2(e) = 2, \ldots, h_k(3) = 4$, *i.e.*, word[1].bit[$h_1(e)$], word[2].bit[$h_2(e)$], ..., word[k].bit[$h_k(e)$]. Then we set the k bits to ones. The example is shown in Figure 2.

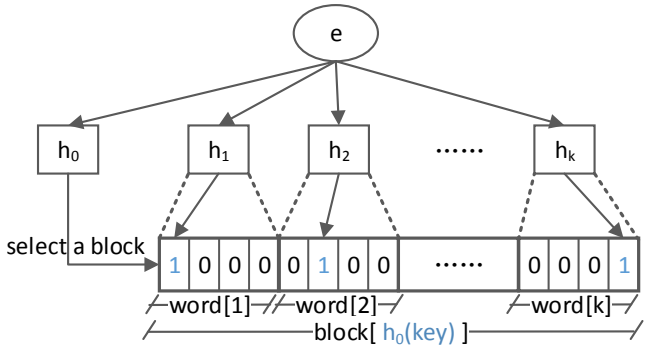


Fig. 2. An example of the insertion process in UFBF.

The check process, checking if a given element belongs to the encoded set, is similar to the insertion process. UFBF first selects a block from the bit array using h_0. Then it selects k bits as the insertion process. If all the k bits are ones, then UFBF returns a positive result (element in the set). Otherwise, it returns a negative result (element not in the set).

B. The Hash Computation Algorithm in UFBF

We develop a novel algorithm for UFBF to calculate the k hash functions in parallel with the use of SIMD instructions. The SIMD instructions are originally designed to accelerate multimedia encoding/decoding. Unless explicitly called, the common programs (by default) do not use these instructions for the consideration of backward-compatibility and cross-platform use. Even with optimization option for some compliers (*e.g.*, the *-O2* option for *gcc*), only a few sentences of the common programs would be complied to SIMD instructions. It is a manner of implicit calls of SIMD instructions.

The hash computation algorithm in UFBF is designed to facilitate the use of SIMD instructions. And we will explicitly call the SIMD instructions to accelerate the hash computation for Bloom filters. Although the SIMD instruction sets are platform-dependent, we use a high-level abstract description of these instructions to introduce our algorithm.

For better understanding, we define two naming rules in our algorithms:

- $vr\_\{\}$ means an SIMD register/variable.
- $v\_\{\}$ means an SIMD operation/command.

Algorithm 1: The hash computation algorithm in UFBF

```
1 seeds[p] ← [seed_1, seed_2, ..., seed_p]
2 hashVals[p] ← [0, 0, ..., 0]
3 vr_seeds ← v_load(seeds)
  /* load the p seeds to an SIMD register   */
4 vr_val ← v_hashFunc(vr_seeds)
  /* implement the SIMD hash function which
     takes p seeds and compute in parallel  */
5 hashVals ← v_store(vr_val)
  /* store the p hash values to memory      */
```

Algorithm 2: The main change from a traditional hash function to its SIMD-version

```
1 val ← val OP a
  /* OP is a general arithmetic operation, val
     stores the intermediate hash value     */
        ⇓
1 vr_a ← v_broadcast(a)
  /* v_broadcast copy p copies of a to vr_a  */
2 vr_val ← v_OP(vr_val, vr_a)
  /* v_OP is the SIMD-version of OP, vr_val
     stores the intermediate p hash values  */
```

Algorithm 3: An example of Algorithm 2 which shows how to translate a traditional addition to SIMD additions in C code

```
1 val = val + a;
  /* val and a are 32-bit(int) integers      */
        ⇓
1 __m256i vr_a = _mm256_set1_epi32(a);
2 __m256i vr_val = _mm256_add_epi32(vr_val, vr_a);
  /* _mm256_set1_epi32, _mm256_add_epi32 are the
     Intel provided API functions for
     AVX/AVX2 instructions                   */
```

Suppose the SIMD instructions can implement p pairs arithmetic operations (*i.e.*, *add*, *sub*, *mul*, *etc*) at the same time, *e.g.*, $(z_1, z_2, \ldots, z_p) = (x_1, x_2, \ldots, x_p) + (y_1, y_2, \ldots, y_p)$. The parameter p is determined by a specific SIMD instruction set. For example, the Intel *SSE* instruction set can implement $p = 4$ pairs of 32-bit arithmetic operations, while the *AVX* instruction sets can implement $p = 8$ pairs of 32-bit arithmetic operations.

The hash computation algorithm in UFBF is shown in Algorithm 1. This algorithm produces p hash values, by using the same hash function with different initial seeds. It first loads p seeds [2] to an SIMD register vr_seeds. Then it uses an SIMD-version hash function *v_hashFunc* to compute the hash values. After it completes the computation, it stores the result from an SIMD register vr_val to memory. The SIMD-version hash function is rewritten with the SIMD instructions according to a traditional hash function. While the SIMD-version hash function is related to a specific traditional hash function, we do not show the algorithm for the *v_hashFunc*. To guide the rewrite rules, we show the main change from a traditional hash function to its SIMD-version in Algorithm 2. For better understanding, we show an example of Algorithm 2 in Algorithm 3, which shows how to translate a traditional addition to SIMD additions in C code. The code follows the Intel provided APIs and could run in CPUs which support AVX/AVX2 instructions. The computation process of a traditional hash function can be summarized as follows: an initial seed encounters a sequence of arithmetic operations, storing each step's result to an intermediate variable. While the SIMD-version can be summarized as that p initial seeds encounter the same sequence of arithmetic operations, storing each step's results to an intermediate SIMD variable. Therefore, the main change of an SIMD-version hash computation is that it has to prepare the SIMD operation data $vr\_val \leftarrow v\_broadcast(a)$ and implement the corresponding SIMD operation v_op.

[2]Many hash functions(*e.g.*, *lookup3* and *murmur* used in this paper) do not have special requirements on the seeds. Therefore we randomly select p seeds, pre-store the seeds in a table and use them when needed.

C. Parallel Bit-Test in Membership Check

In the membership check process, standard Bloom filters (and their variants) must test k bits sequentially, *i.e.*, test k bits one by one and return *negative* once encountered a zero-bit. If zero-bit is not encountered at the end of this bit-test process, return *positive*. In our UFBF, we change the *sequential* bit-test process to *parallel* bit-test process, reducing the complexity from $O(k)$ to $O(1)$. In order to achieve *parallel* bit-test, we make two improvements for the membership check.

First, we change the bloom filter data structure to a *block-word* style (shown in Figure 1), which brings in parallelism for membership check. The parallelism is reflected in that the UFBF encodes an element to k consecutive words (in a block), and these words can be fetched and tested at the same time. As the standard Bloom filter encodes an element to k arbitrary locations of the bit array, it does not have this kind of parallelism, for the reason that it has to fetch a bit (from memory to register) and test it for every membership bit in the bit array. Second, we develop a new membership check algorithm, which implements a parallel bit-test process with the aid of SIMD instructions. This algorithm computes the k hash functions in parallel, effectively reducing hash computation time.

The membership check algorithm for UFBF is shown in Algorithm 4. In this algorithm, the k membership bits are tested in parallel. The k hash functions for membership check in this algorithm are calculated in parallel using Algorithm 1. Actually, we make an assumption here that $k \leq p$, which means the p hash values produced by Algorithm 1 can satisfy the Bloom filter's hash function need. Since p is a fixed parameter for a specific SIMD instruction set, we can not set k an arbitrary value in practice. This problem can be solved by an extension of UFBF, introduced in Section IV.

Algorithm 4: The membership check algorithm in UFBF

```
Function membershipCheck(element e)
    loc ← compute the block index of e
    vr_val ← compute k hash values using Algorithm 1
    vr_a ← v_broadcast(1)
    vr_a ← v_shiftLeft(vr_a, vr_val)
    /* v_shiftLeft shifts k words in vr_a
       left in parallel by the amount
       specified by vr_val                    */
    vr_b ← v_load(&block[loc])
    vr_b ← v_not(vr_b)
    /* v_not is bitwise NOT operation         */
    v_test(vr_a, vr_b)
    /* v_test is bitwise AND operation        */
    if zero-flag is set then
        return positive
    end
    return negative
end
```

D. Cache Efficiency for Membership Check

The cache efficiency for membership check of UFBF is expected to be far better than SBF. In UFBF, an element's information is encoded in a small block of the bit array , and a block can easily fit into one cache-line of CPU's cache. In SBF, an element's information is encoded in k arbitrary locations of the bit array, and at most k cache misses could occur during one membership check process.

Formally, we analyze the worst case cache misses in one membership check of UFBF. In fact, we make an assumption in the following proofs: once a cache miss occurs, the CPU would load the corresponding cache-line immediately, from off-chip memory (or, low-level cache) into on-chip cache (or, high-level cache). This assumption is true in practice for most of the CPUs.

In Theorem 1, we prove that two cache misses would occur in the worst case in one membership check process, if the block size is no more than the cache-line size. While this constraint can be easily satisfied in practice, we want to reduce (at most) two cache misses to (at most) one cache miss in one membership check.

In Theorem 2, we prove that only one cache miss would occur in the worst case in one membership check process, if a more stringent condition is satisfied, *i.e.*, the block size divides the cache-line size and the bit array is cache-line size aligned. In practice, the cache-line usually has size a power of 2, which means the block size should also be a power of 2 (Corollary 1). By Corollary 2, the hash function number k should be a power of 2, which suggests that $k = 2, 4, 8, 16$ *etc*. That is to say, UFBF experiences better cache efficiency when k is a power of 2 than when k is other values.

Theorem 1. *Suppose the cache-line size is L. If the block size satisfies $b \leq L$, at most two cache misses would occur in one membership check.*

Proof by Contradiction. Suppose that more than two cache misses occur during one membership check. Because an element's information is encoded in just one block in UFBF, only one block memory would be missed in cache during one membership check. Denote the starting and end address of the missed block as $addr\_s$, $addr\_e$, respectively. The supposition, more than two cache misses, means $\exists\ t \in \mathbb{N}$ such that $addr\_s < tL < (t+1)L < addr\_e$. Then we can get $b = addr\_e - addr\_s > [(t+1)L - tL] = L$, which contradicts $b \leq L$ in the statement. □

Theorem 2. *Suppose the cache-line size is L. If the block size satisfies $b|L$ and the bit array is L-aligned, at most one cache miss would occur in one membership check.*

Proof by Contradiction. Suppose that more than one cache miss occurs during one membership check. Denote the starting and end address of the missed block as $addr\_s$, $addr\_e$, respectively. Because $b|L$ and the bit array is L-aligned, then $\exists\ s, i, j \in \mathbb{N}$ such that $L = sb$, $addr\_s = iL + jb$, $addr\_e = iL + (j+1)b$. The supposition, more than one cache miss, means $\exists\ t \in \mathbb{N}$ such that $addr\_s < tL < addr\_e$. Substitute $addr\_s$, $addr\_e$, we get $iL+jb < tL < iL+(j+1)b$. Further, we get $isb+jb < tsb < isb+(j+1)b$. Simplify this formula, we can get $0 < (t-i)s + j < 1$. Apparently, $(t-i)s + j$ is an integer and we get a contradiction. □

Corollary 1. *Suppose the cache-line size L is a power of 2. Then we can conclude that if $b \leq L$, b is a power of 2, and the bit array is L-aligned, at most one cache miss would occur in one membership check.*

Corollary 2. *Suppose w is a power of 2. Then we can conclude that if $k = \frac{b}{w} \leq \frac{L}{w}$, k is a power of 2, and the bit array is L-aligned, at most one cache miss would occur in one membership check.*

E. False Positive Probability Analysis

The false positive probability of our proposed UFBF, f_u, is analyzed as follows. Assume we use fully random hash functions. Let $\mathcal{F}$ be the false positive event that an element e', which is not in the set, is mistakenly regarded as in the set. To check the membership of e', it is hashed to k words, each word is set one bit, in a membership block. Suppose x elements have been inserted to this membership block, where $x \in [0, n]$. Then a bit is set in a word with probability $1-(1-\frac{1}{w})^x$. Let $\mathcal{X}$ be the random variable that represents how many elements have been inserted to a block. Then the conditional probability for $\mathcal{F}$ to occur when $\mathcal{X} = x$ is:

$$Pr\{\mathcal{F}|\mathcal{X} = x\} = \left(1 - \left(1 - \frac{1}{w}\right)^x\right)^k \tag{4}$$

Obviously, $\mathcal{X}$ follows the binomial distribution, $Bino(n, \frac{1}{r})$, then we can get

$$Pr\{\mathcal{X} = x\} = \binom{n}{x}\left(\frac{1}{r}\right)^x\left(1 - \frac{1}{r}\right)^{n-x}, \forall\ 0 \leq x \leq n \tag{5}$$

Then, we can get the false positive probability of UFBF as:

$$f_u = Pr\{\mathcal{F}\} = \sum_{x=0}^{n}(Pr\{\mathcal{X}=x\}\cdot Pr\{\mathcal{F}|\mathcal{X}=x\}) = \sum_{x=0}^{n}\binom{n}{x}\left(\frac{1}{r}\right)^x\left(1-\frac{1}{r}\right)^{n-x}\left(1-\left(1-\frac{1}{w}\right)^x\right)^k \quad (6)$$

In the previous analysis, we do not show the effect of hash function h_0 that is used to select the block. Usually the hash values produced by Algorithm 1 would have unused hash bits which can be utilized by h_0. However, if the unused hash bits can not satisfy the requirement of h_0, then the hash computation cost of h_0 should be considered.

It is difficult to compare the false positive probability of UFBF and SBF directly using equations, therefore we make some numerical calculations to find the trend. Table II presents the comparison of theoretical false positive probability between SBF (f_s) and UFBF (f_u). It can be concluded from this table that UFBF has higher false positive probability than SBF. In other words, UFBF needs to use more memory to achieve the same false positive probability as SBF. We find that the false positive probability changes intensely for small load factors ($\frac{n}{m}$) both for UFBF and SBF. When the load factor of Bloom filters increases, the difference of false positive probability between UFBF and SBF ($\frac{f_u-f_s}{f_s}$) decreases. We also find that $\frac{f_u-f_s}{f_s}$ nearly halves if the word size (w) of UFBF changes from 32 to 64. From the perspective of false positive probability, we prefer larger word size for UFBF. However, the word size usually is restricted by the SIMD instructions CPU supported.

TABLE II
COMPARISON OF THE THEORETICAL FALSE POSITIVE PROBABILITY BETWEEN SBF AND UFBF, $n = 10000, k = 4$

load factor (n/m)	f_s	$w=32$		$w=64$	
		f_u	$\frac{f_u-f_s}{f_s}$	f_u	$\frac{f_u-f_s}{f_s}$
0.02	3.49 e-5	1.39 e-4	2.98	7.98 e-5	1.28
0.04	4.78 e-4	1.02 e-3	1.14	7.35 e-4	0.54
0.06	2.07 e-3	3.44 e-3	0.66	2.73 e-3	0.32
0.08	5.62 e-3	8.11 e-3	0.44	6.85 e-3	0.22
0.10	1.18 e-2	1.56 e-2	0.32	1.37 e-2	0.16
0.12	2.11 e-2	2.62 e-2	0.24	2.37 e-2	0.12
0.14	3.38 e-2	4.01 e-2	0.19	3.70 e-2	0.09
0.16	4.99 e-2	5.75 e-2	0.15	5.37 e-2	0.08
0.18	6.94 e-2	7.78 e-2	0.12	7.36 e-2	0.06
0.20	9.20 e-2	1.01 e-1	0.10	9.69 e-2	0.05

F. Discussion

In general, the UFBF attempts to improve membership check performance by introducing parallel operations in membership check process and using SIMD instructions to accelerate these parallel operations. The SIMD instructions are well supported by general and embedded CPUs. However, the dedicated hardware like core router which uses hardware forwarding engines or network processors which do not supply the SIMD instructions, this advanced feature cannot be utilized. Another issue of the UFBF is the poor scalability of hash function number for indexing membership bits. Due to restriction of underlying SIMD instructions, the UFBF can only support no more than p hash functions, where p is determined by a specific SIMD instruction set. Although this issue can be relieved by using more powerful instruction sets with larger parallelism, in next section, we introduce a generalization of UFBF, which addresses this issue completely.

IV. A GENERALIZATION OF UFBF

The UFBF does not support large hash function number k, since k is restricted by the underlying SIMD instructions. In this section, we introduce a generalization of UFBF, called c-UFBF, which has scalability for the number of hash functions. The c-UFBF has the same basic data structure with UFBF, as is shown in Figure 1. The difference is that, when inserting an element, c-UFBF randomly selects c blocks to encode an element. The operation for each selected block is the same as UFBF.

A. False Positive Probability of c-UFBF

Let k be the total bits used for membership check. Let $k_1, k_2, \ldots, k_c$ be the bits used for each block's membership check. We have $k = \sum_{i=1}^{c} k_i$. To simplify the false positive analysis of c-UFBF, we assume each block has equal number of membership bits $k_i = \frac{k}{c}, i \in [1, c]$.

Let f_c be the false positive probability of c-UFBF. Assume an element e' is not in the set. To check the membership of e', c blocks are selected. Let us first analyze the event $\mathcal{F}$ that $\frac{k}{c}$ bits in one of the c blocks are all ones. Let $\mathcal{X}$ be the random variable that represents how many times a block has been selected to encode an element. Suppose x is a specific value of $\mathcal{X}$, where $x \in [0, nc]$. Then the conditional probability for $\mathcal{F}$ to occur when $\mathcal{X} = x$ is:

$$Pr\{\mathcal{F}|\mathcal{X}=x\} = \left(1-\left(1-\frac{1}{w}\right)^x\right)^{k/c} \quad (7)$$

As each element selects c blocks to encode its membership, a total of nc selections are made in the insertion process. Assume we use fully random hash functions. Hence, a specific block will be selected with probability $\frac{1}{r}$ in each selection. Therefore, $\mathcal{X}$ follows the binomial distribution, $Bino(nc, \frac{1}{r})$, then we can get:

$$Pr\{\mathcal{X}=x\} = \binom{nc}{x}\left(\frac{1}{r}\right)^x\left(1-\frac{1}{r}\right)^{nc-x} \quad (8)$$

The probability for $\mathcal{F}$ to happen is:

$$Pr\{\mathcal{F}\} = \sum_{x=0}^{n}(Pr\{\mathcal{X}=x\}\cdot Pr\{\mathcal{F}|\mathcal{X}=x\}) = \sum_{x=0}^{cn}\binom{cn}{x}\left(\frac{1}{r}\right)^x\left(1-\frac{1}{r}\right)^{cn-x}\left(1-\left(1-\frac{1}{w}\right)^x\right)^{\frac{k}{c}} \quad (9)$$

The element e' selects c blocks for membership check. If the event $\mathcal{F}$ happens in all these blocks, a false positive happens

in c-UFBF. Therefore, the false positive probability of c-UFBF is:

$$f_c = (Pr\{\mathcal{F}\})^c = \left[\sum_{x=0}^{cn}\binom{cn}{x}\left(\frac{1}{r}\right)^x\left(1-\frac{1}{r}\right)^{cn-x}\left(1-\left(1-\frac{1}{w}\right)^x\right)^{\frac{k}{c}}\right]^c \tag{10}$$

Obviously, the $f_{c|c=1} = f_u$. When $c = k$, we can get:

$$\begin{aligned} f_{c|c=k} &= \left[\sum_{x=0}^{kn}\binom{kn}{x}\left(\frac{1}{r}\right)^x\left(1-\frac{1}{r}\right)^{kn-x}\left(1-\left(1-\frac{1}{w}\right)^x\right)\right]^k \\ &= \left[1-\sum_{x=0}^{kn}\binom{kn}{x}\left(\frac{1}{r}\left(1-\frac{1}{w}\right)\right)^x\left(1-\frac{1}{r}\right)^{kn-x}\right]^k \\ &= \left[1-\left(\frac{1}{r}\left(1-\frac{1}{w}\right)+1-\frac{1}{r}\right)^{kn}\right]^k \\ &= \left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^k = f_s \end{aligned} \tag{11}$$

The numerical result of false positive probability of c-UFBF is shown in Table III. When the parameter c increases from 1 to 2 and 4, the false positive probability of c-UFBF presents a decreasing trend ($f_{c|c=1} > f_{c|c=2} > f_{c|c=4}$). Due to $f_{c|c=1} = f_u, f_{c|c=4} = f_s$, we can conclude from this table that $f_u \geq f_c \geq f_s$. According to our calculations, this decreasing trend is held for all common parameter settings of c-UFBF. That is to say, the false positive probability of c-UFBF is between UFBF and SBF ($f_u \geq f_c \geq f_s$). When the load factor ($\frac{n}{m}$) increases, the false positive probability of c-UFBF presents a decreasing trend, which is similar to UFBF.

TABLE III
COMPARISON OF THE THEORETICAL FALSE POSITIVE PROBABILITY FOR C-UFBF WHEN c CHANGES, $n = 10000, w = 32, k = 4$. IN THIS TABLE, $f_1 = f_{c|c=1}, f_2 = f_{c|c=2}, f_4 = f_{c|c=4}$

load factor (n/m)	$c=1$	$c=2$		$c=4$	
	f_1	f_2	$\frac{f_2-f_1}{f_1}$	f_4	$\frac{f_4-f_1}{f_1}$
0.02	1.39 e-4	6.47 e-5	−0.53	3.49 e-5	−0.75
0.04	1.02 e-3	6.50 e-4	−0.36	4.78 e-4	−0.53
0.06	3.44 e-3	2.52 e-3	−0.27	2.07 e-3	−0.40
0.08	8.11 e-3	6.45 e-3	−0.20	5.62 e-3	−0.31
0.10	1.56 e-2	1.31 e-2	−0.16	1.18 e-2	−0.24
0.12	2.62 e-2	2.28 e-2	−0.13	2.11 e-2	−0.19
0.14	4.01 e-2	3.60 e-2	−0.10	3.38 e-2	−0.16
0.16	5.75 e-2	5.26 e-2	−0.08	4.99 e-2	−0.13
0.18	7.78 e-2	7.23 e-2	−0.07	6.94 e-2	−0.11
0.20	1.01 e-1	9.52 e-2	−0.06	9.20 e-2	−0.09

B. Membership Check Overhead of c-UFBF

It is easy to prove that c-UFBF has c times the membership check overhead of UFBF. The increased overhead is threefold. First, it will cause $2c$ cache misses in the worst case for one element membership check. If alignment is satisfied as UFBF, it will cause c cache misses in the worst case for one element membership check. Second, it will take c times of the hash computation time used by UFBF. Third, it will take c times of the bit-test time used by UFBF. Therefore, we should minimize c if we aim to build fast Bloom filters. Due to $1 \leq c \leq k$, the membership check overhead of c-UFBF is between UFBF and SBF (assuming the membership check overhead of UFBF is just $\frac{1}{k}$ of SBF).

C. Discussion

The c-UFBF extends the UFBF to support a larger number of hash functions. Instead of selecting one block to encode an element as UFBF, c-UFBF selects c blocks to encode an element. Though c-UFBF has better scalability of hash function number, it has c times the membership check overhead of UFBF. For not sacrificing the membership check performance a lot, a small c should be employed in c-UFBF. Actually, the c-UFBF is a tradeoff between UFBF and SBF. With the same memory requirement, UFBF has higher false positive probability and lower membership check overhead than SBF. While c-UFBF's false positive probability and membership check overhead are both between UFBF and SBF.

V. EVALUATION

We make experiments to evaluate our proposed UFBF and its generalization c-UFBF.

A. Experiment Setup

Platform: We implement the experiments on a commodity server with Intel CPU Core i7-4790 (4 cores × 2 threads, 3.6 GHz). Each core of this CPU has independent L1 cache (L1 D-Cache is 32 KBytes, L1 I-Cache is 32 KBytes) and L2 cache (256 KBytes). The 4 cores share L3 Cache (8 MBytes). The cache-line size is 64-byte (512 bits). This server has 16GB DDR3 (1600 MHz) memory. This server runs Microsoft Windows 7, 64-bit operating system.

SIMD instructions: The Intel i7-4790 CPU supports several SIMD instruction sets. We use the *AVX, AVX2* instruction sets in our experiments. Because *AVX2* is a simple extension of *AVX*, we use the term *AVX* to represent *AVX, AVX2* in the following description if there is no confusion. These two instruction sets can operate 16 256-bit registers [19]. *AVX* can implement eight 32-bit signed/unsigned integer arithmetic operations in parallel. Most of the *AVX* SIMD instructions could be called using C/C++ style functions provided by Intel Intrinsics Guide [26]. We use C/C++ programming language to code our evaluation programs. The complier we use is *gcc*. To use the *AVX, AVX2* instruction sets, the special options *-mavx, -mavx2* are needed for *gcc*.

Datasets: We use the real-world Internet traces, obtained from CAIDA [27], to evaluate the performance of UFBF. The trace is extracted from a backbone 10Gbps link and lasts 60 minutes. It contains 2G IPv4 packets, 5M different destination IP addresses, and 50M flows (flow identifier is <*srcIP, dstIP, srcPort, dstPort, protocol*>). We use two datasets extracted from the traces for our evaluation, as shown in Table IV.

TABLE IV
DATASETS USED IN THE FOLLOWING EXPERIMENTS

name	data	ID length	# of items
dataset1	dst IPs	4 bytes	5 M
dataset2	flows	13 bytes	50 M

B. The Hash Computation Evaluation

To test the performance of the hash computation algorithm in UFBF (Algorithm 1), we make two comparative experiments. We use the traditional hash functions *murmur* [28] and *lookup3* [29] as the compared hash functions. We set the hash value's bit-width as 32-bit. Since the *AVX* instruction set uses 256-bit registers, the hash computation algorithm in UFBF can compute (at most) 8 hash functions in parallel.

Figure 3 and Figure 4 show the evaluation results. We can find that the *lookup3* hash function consumes 23 clock cycles on average for one hash computation. As the computation time is proportional to the hash function number, a linear increasing trend occurs for computing more hash functions. However, the SIMD-version (using Algorithm 1) implementation of *lookup3* has a constant computation time when hash function number ranges from 1 to 8. We find that *lookup3-SIMD* consumes 1.78 times the time of *lookup3* for computing one hash function. This difference comes from two aspects. First, the SIMD-version hash function has to use additional instructions to prepare the data for SIMD registers. Second, an SIMD instruction usually takes slightly more time compared to a corresponding common instruction. The slightly increased time for computing one hash function can be compensated when the hash function number increases. We can conclude that more hash functions used, the more time we can reduce for SIMD-version hash functions. The comparison of *murmur* and its SIMD-version has a similar result with *lookup3*. However, the *murmur-SIMD* consumes 2.43 times the time of *murmur* for computing one hash function. The larger ratio (2.43>1.78) comes from that *murmur* uses many integer multiplications, but the *AVX* instruction set has a relatively poor support for vector integer multiplication.

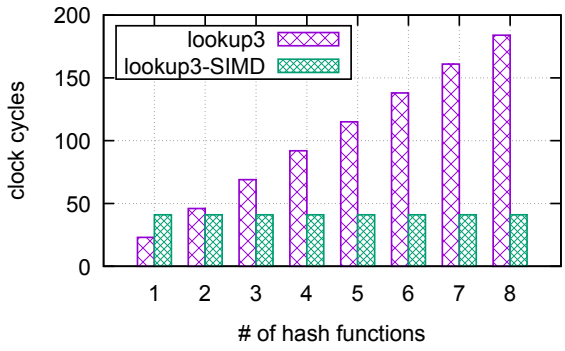


Fig. 3. The run time comparison between *lookup3* hash function and its SIMD-version.

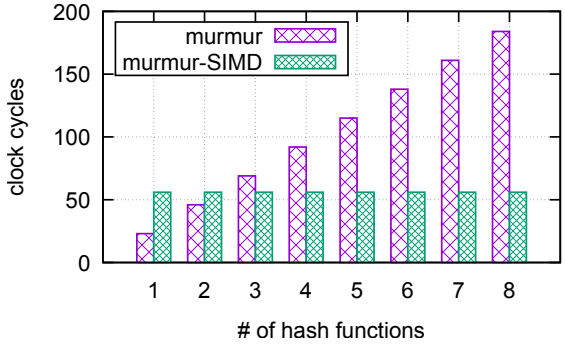
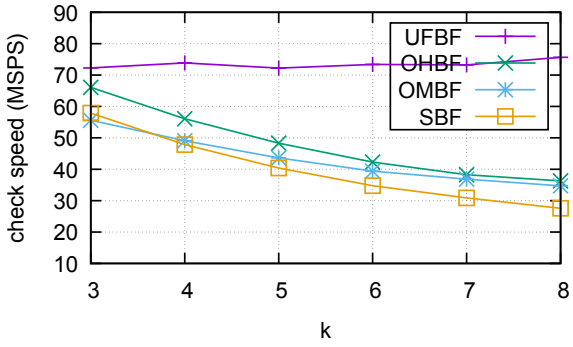


Fig. 4. The run time comparison between *murmur* hash function and its SIMD-version

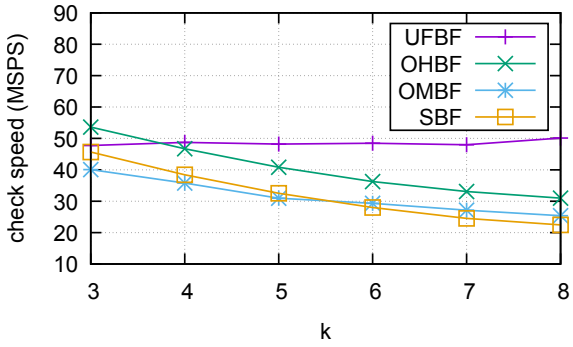
C. Membership Check Speed Evaluation

We compare three Bloom filters, SBF [20], OMBF [22], and OHBF [24], with our proposed UFBF. OMBF and OHBF are two state-of-the-art Bloom filter variants which attempt to reduce the membership check overhead of Bloom filters. OMBF attempts to reduce the memory overhead, while OHBF attempts to reduce the hash computation overhead. UFBF makes improvements to reduce both memory overhead and hash computation overhead at the same time. By using SIMD instructions, UFBF can achieve parallel bit-test in membership check, which reduces the complexity of bit-test process from $O(k)$ to $O(1)$. With these advantages, UFBF is expected to improve the membership check speed effectively. We use two datasets listed in Table IV to conduct our experiments. In the experiments, *negative check* means checking an element not in the set, while *positive check* means checking an element in the set. We use MSPS (Millions Searches Per Second) as the unit of membership check speed in the following experiments. The bit array of Bloom filters is cache-line size aligned. The inserted items (that are encoded to Bloom filters, called encoded-set) are selected randomly from the two datasets (5M and 50M). The queries are selected differently for positive-check and negative-check experiments. In the positive-check experiments, we repeat lookup the items in the encoded-set several times. In the negative-check experiments, the queries are selected randomly from items which are not in the encoded-set.

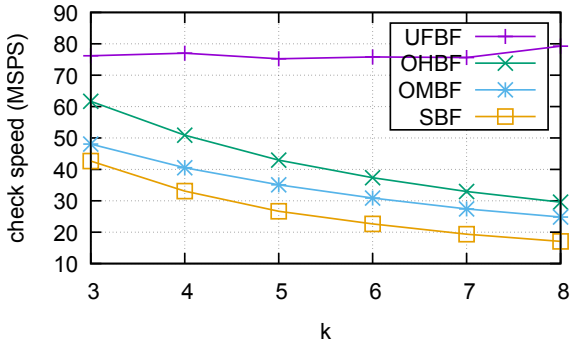
Figure 5 presents the membership check speed comparison of the four Bloom filters, SBF, OMBF, OHBF, and UFBF. We can see that the check speed of SBF, OMBF, and OHBF shows an decreasing trend. This is because SBF, OMBF, and OHBF implement membership check using a sequential bit-test process, and there are more membership bits to check on average with the growth of k. While our UFBF presents a (nearly) constant check speed in these experiments due to parallel hash computation and parallel bit-test in membership check. The small jitters of check speed in UFBF comes from the different cache efficiency for different values of k (discussed in Section III-D). For SBF, OMBF, and OHBF, positive check has more membership bits to check (and more hash computations correspondingly) on average than negative check, therefore positive check is slower than negative check.



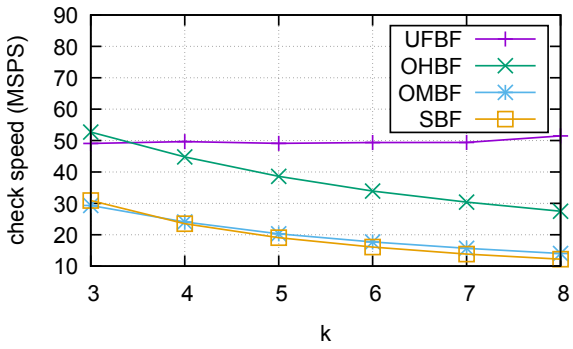
(a) dataset1, negative check



(b) dataset2, negative check



(c) dataset1, positive check



(d) dataset2, positive check

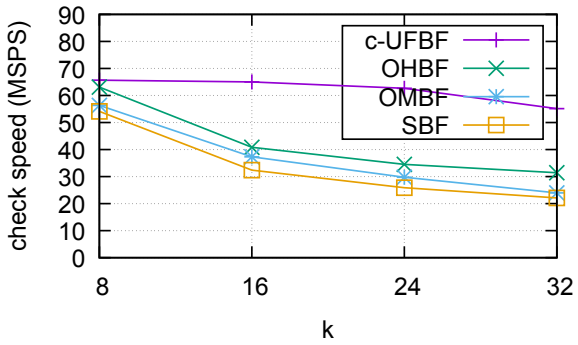
Fig. 5. The membership check speed of the four Bloom filters: SBF, OMBF, OHBF, and UFBF. MSPS stands for millions searches per second. We set $n = 10^5, m = 10^6$. The load factor is $\frac{n}{m} = 0.1$. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.

For UFBF, positive check and negative check both test all membership bits in parallel, therefore positive check and negative check have (nearly) the same speed. As the CPU's cache size (64 Mbits) is larger than Bloom filer bit arrays' size (1 Mbits) and OMBF mainly aims to improve the cache efficiency of Bloom filters, OMBF only has slightly faster check speed than SBF in these experiments. As OHBF reduces the hash computation overhead a lot and cache efficiency is not a big issue in these experiments, OHBF has faster membership check speed than SBF and OMBF. While UFBF takes both hash computation overhead and cache efficiency into consideration and improves the bit-test speed, UFBF has the fastest membership check speed in the four Bloom filer variants. When $k = 8$, UFBF doubles the membership check speed than SBF in negative check, and it has four times the membership check speed of SBF in positive check.

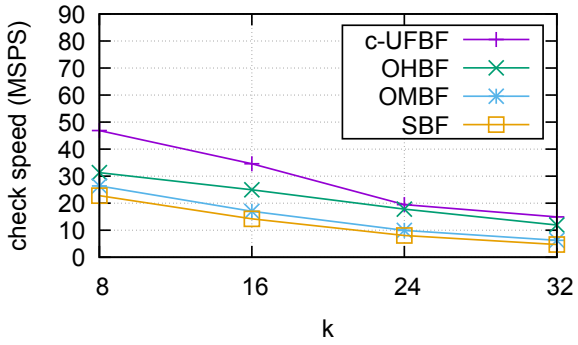
Figure 6 presents the membership check speed comparison of the four bloom filters, SBF, OMBF, OHBF and c-UFBF. We can see that the membership check speed of the four bloom filters shows an decreasing trend. For OHBF, OMBF and SBF, the lower check speed has been explained previously. For c-UFBF, when k is greater than 8, c-UFBF has to perform extra SIMD instructions to conduct the membership check process. However, due to parallel hash computation and parallel bit-test, c-UFBF shows the highest membership check speed than the other three bloom filters when a large k is set.

Figure 7 presents the negative and positive membership check speed comparison of the four Bloom filters, SBF, OMBF, OHBF, and UFBF, when m varies. The L1, L2, and L3 CPU cache sizes are annotated. We can see that the membership check speed of the four Bloom filters is nearly constant when the CPU has adequate cache ($m <$ L2-Cache), and it drops slowly when the CPU cache is not so adequate (L2-Cache $< m <$ L3-Cache). The membership check speed drops quickly when the CPU cache is not enough ($m >$ L3-Cache) to accommodate the bit array. As UFBF improves the cache efficiency in its design, it outperforms the other three Bloom filters on membership check speed, whether the on-chip memory is enough to accommodate the bit array or not. Since OHBF's design does not consider the cache efficiency, its membership check speed drops sharply when the bit array size (m) approaches the size of L3-Cache.

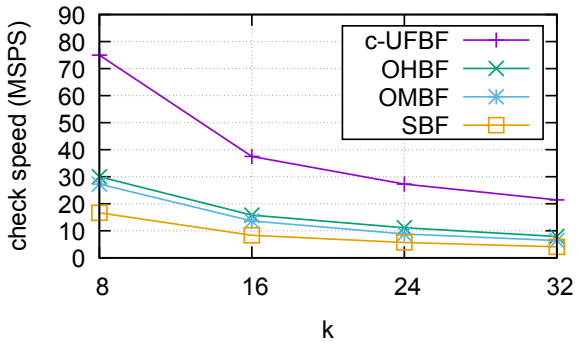
Figure 8 presents the membership check speed comparison of c-UFBF with different configurations. For a same dataset, the positive check and negative check have different performance trends when the load factor increases. In positive check, the check speed is (nearly) constant when c is fixed and load factor varies. Another feature in positive check is that the check speed decreases when c increases. The reason is that the membership check overhead is proportional to c in c-UFBF, as discussed in Section IV-B. In negative check, the check speed is (nearly) constant when $c = 1$, but the check speed decreases when $c = 2, 3$ and the load factor increases. When $c = 1$, the c-UFBF is equal to UFBF, it only has to check just one block to make sure whether an element is in the set or not. However, when $c > 1$, c-UFBF has to check the c blocks one by one and stop the search process once a block's membership bits are not all ones. So the check speed is inversely proportional



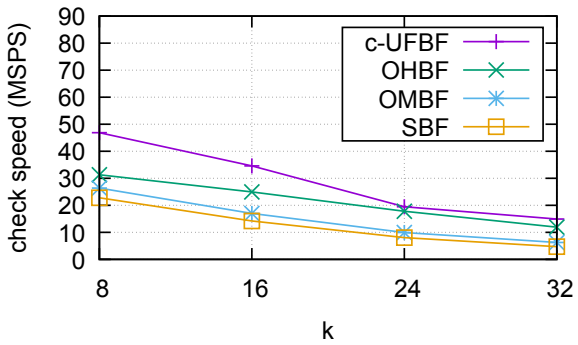
(a) dataset1, negative check



(b) dataset2, negative check

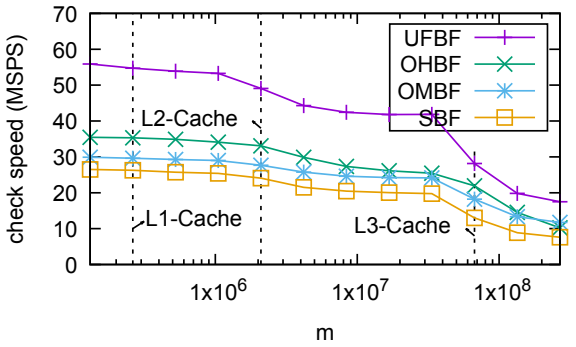


(c) dataset1, positive check

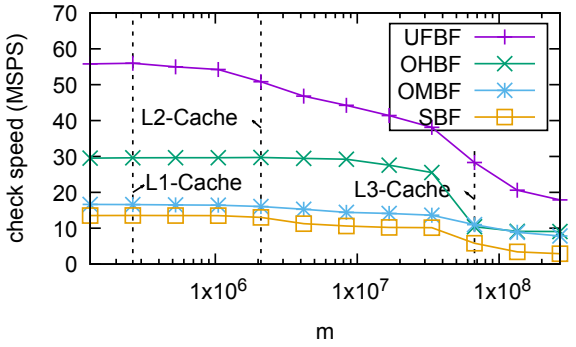


(d) dataset2, positive check

Fig. 6. The membership check speed of the c-UFBF when k is set to different values. We set $m = 10^6$, $\frac{k}{c} = 8$. The load factor is $\frac{n}{m} = 0.04$. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.



(a) negative check



(b) positive check

Fig. 7. The membership check speed of the four Bloom filters: SBF, OMBF, OHBF, and UFBF. We set $k = 8$. The load factor is $\frac{n}{m} = 0.1$. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.

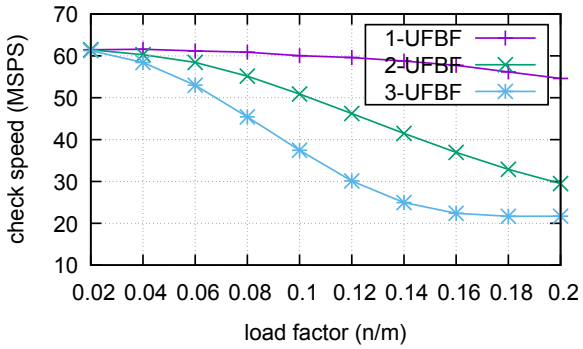
to the average number of blocks needed to access. Obviously, the average number of blocks needed to access increases with the growth of load factor, as more ones are inserted to the bit array.

D. False Positive Evaluation

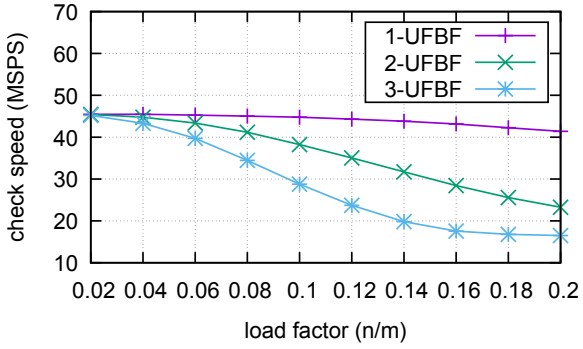
We make two experiments on false positive ratio evaluation. Figure 9 presents the comparison on false positive ratio between theory and simulation results of UFBF. We can see that the two simulation results on two datasets exactly match the theoretical analysis, which validates the false positive probability analysis of UFBF in Section III-E. Figure 10 presents the comparison on false positive ratio between theory and simulation results of c-UFBF when $c = 2$. Again the the two simulation results on two datasets exactly match the theoretical analysis, which validates the false positive probability analysis of c-UFBF in Section IV-A.

VI. Conclusions

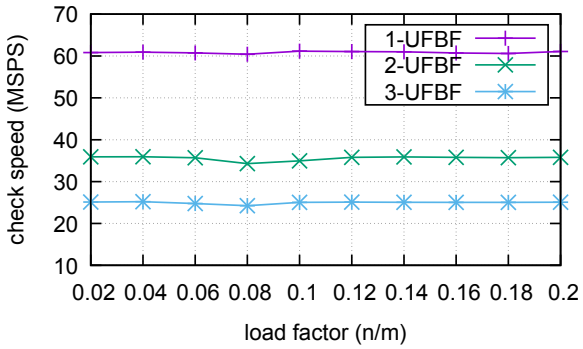
In recent years, Bloom filters have been widely used in all aspects of the network applications due to their simplicity and efficiency. However, with the rapid development of network technology, increasingly strict requirements on the network speed and latency are put forward, which goes beyond the ability of traditional Bloom filters. In this paper, we propose a new Bloom filter variant called Ultra-Fast Bloom Filter,



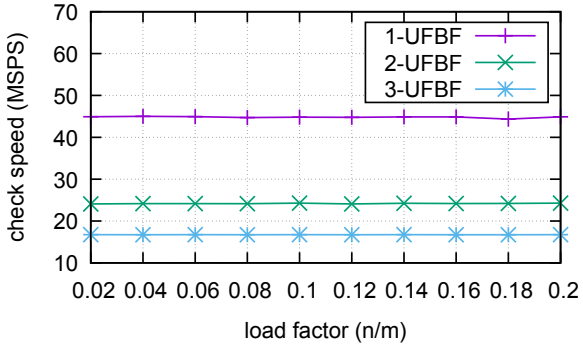
(a) dataset1, negative check



(b) dataset2, negative check



(c) dataset1, positive check



(d) dataset2, positive check

Fig. 8. The membership check speed of the c-UFBF when c is set to different values. We set $m = 10^6$, $\frac{k}{c} = 4$. The load factor is $\frac{n}{m} = 0.1$. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.

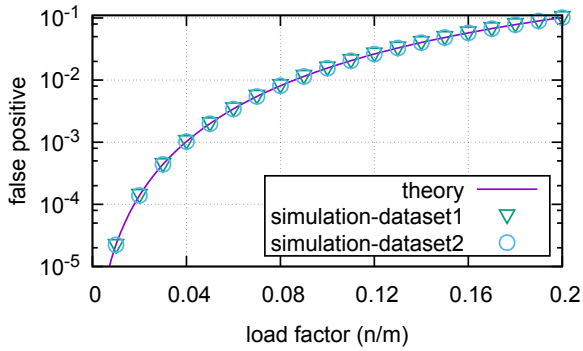


Fig. 9. The false positive ratio of UFBF, $n = 10000, w = 4, k = 4$. Each point in this figure is the mean of 1,000 experiments.

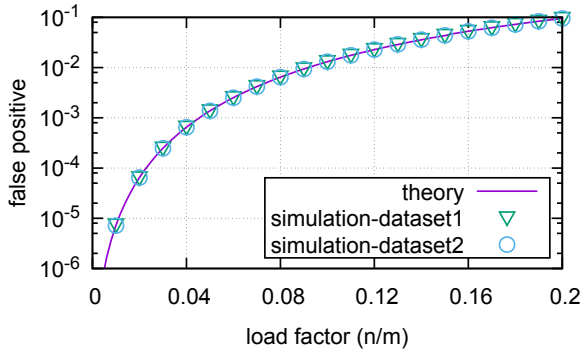


Fig. 10. The false positive ratio of c-UFBF when $c = 2, n = 10000, w = 4, k = 4$. Each point in this figure is the mean of 1,000 experiments.

which has significant advantages over traditional Bloom filters in three key factors, *i.e.*, hash computation, membership bit-tests, and cache efficiency. We develop a novel hash computation algorithm, which can compute the hash functions in parallel with the use of SIMD instructions. Again, by the use of SIMD instructions, the traditional sequential bit-test process is changed to parallel bit-test process. Since SIMD instructions are widely supported by most of the modern CPUs, our UFBF design has a very good application prospect. The UFBF also has good cache efficiency as it encodes an element's information to a small block which can easily fit into a cache-line. Numerical results show that the UFBF has a higher false positive rate in most of the settings. However, compared to its improvement in performance, the tradeoff is absolutely worthwhile. Further, we introduce a generalization of UFBF, called c-UFBF, which has better scalability in terms of the number of hash functions. Actually, the c-UFBF is a tradeoff between UFBF and SBF. Either in terms of the false positive probability or the membership check overhead, the performance of c-UFBF is between UFBF and SBF.

ACKNOWLEDGEMENTS

This work is sponsored by Huawei Innovation Research Program (HIRP), NSFC (61373143, 61432009, 61872213).

REFERENCES

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

[2] 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.

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

[4] 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.

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

[6] F. Chang, W.-c. Feng, and K. Li, "Approximate caches for packet classification," in *IEEE INFOCOM*, 2004.

[7] 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.

[8] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond bloom filters: from approximate membership checks to approximate state machines," in *ACM SIGCOMM Computer Communication Review*, 2006.

[9] Y. Yao, S. Xiong, J. Liao, M. Berry, H. Qi, and Q. Cao, "Identifying frequent flows in large datasets through probabilistic bloom filters," in *IEEE IWQoS*, 2015.

[10] 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.

[11] H. Dai, Y. Wang, H. Wu, J. Lu, and B. Liu, "Towards line-speed and accurate on-line popularity monitoring on ndn routers," in *IEEE IWQoS*, 2014.

[12] 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.

[13] "Huawei launches world's first end-to-end 100g solutions," http://pr.huawei.com/en/news/hw-062645-corporate-ran-wnm-ran-wnp-ds-wisg-vs-win.htm#.WKgFdtV96Uk, 2017.

[14] "Cisco crs-x," http://www.cisco.com/c/en/us/products/routers/carrier-routing-system/index.html, 2017.

[15] "Huawei ne9000," http://e.huawei.com/en/products/enterprise-networking/routers/ne/ne9000, 2017.

[16] "Crypto++ 5.6.0 benchmarks," http://www.cryptopp.com/benchmarks.html, 2017.

[17] M. Mitzenmacher and S. Vadhan, "Why simple hash functions work: Exploiting the entropy in a data stream," in *SODA*, 2008.

[18] 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.

[19] "Intel 64 and ia-32 architectures software developer manuals," https://software.intel.com/en-us/articles/intel-sdm, 2017.

[20] 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.

[21] O. Polychroniou and K. A. Ross, "Vectorized bloom filters for advanced simd processors," in *Proceedings of International Workshop on Data Management on New Hardware*, 2014.

[22] Y. Qiao, T. Li, and S. Chen, "One memory access bloom filters and their generalization," in *IEEE INFOCOM*, 2011.

[23] F. Putze, P. Sanders, and J. Singler, "Cache-, hash- and space-efficient bloom filters," in *International Workshop on Experimental and Efficient Algorithms*, 2007.

[24] J. Lu, T. Yang, Y. Wang, H. Dai, L. Jin, H. Song, and B. Liu, "One-hashing bloom filter," in *IEEE IWQoS*, 2015.

[25] 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.

[26] "Intel intrinsics guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2017.

[27] C. Walsworth, E. Aben, kc claffy, and D. Andersen, "The caida anonymized internet traces," 2016, http://www.caida.org/data.

[28] A. Appleby, "Murmurhash3," https://github.com/aappleby/smhasher/tree/master/src, 2017.

[29] B. Jenkins, "lookup3 hash function," http://www.burtleburtle.net/bob/c/lookup3.c, 2017.

Jianyuan Lu is a joint postdoctoral research fellow in Tsinghua University and Alibaba Cloud. He received the B.S. degree in Information and Computing Science from Beijing University of Posts and Telecommunications, Beijing, China, in 2011, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2017. His research interests include cloud computing, software defined networking(SDN), network measurements, high performance network algorithm, and power-proportional network design.

Ying Wan received the B.S. degree in Communication Engineering from Northwestern Polytechnical University in 2016. He is currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include high performance network algorithm, and software defined network.

Yang Li received the B.S. degree in Communication Engineering from Sichuan University, Chengdu, China, in 2012. He is currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include network measurements, software defined networking and named data networking.

Chuwen Zhang Received the B.S. degree in communication engineering from Northwestern Polytechnical University, Xian, China, in 2015. He is Currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include high-performance switches/routers, named data networking and vehicle networks.

Huichen Dai is currently a Senior Engineer at Huawei Technologies Co., Ltd. He is now working on congestion control algorithms for RDMA. Prior to this, he was a Post-Doctoral Research Fellow with the Department of Computer Science and Technology, Tsinghua University. He received the B.S. degree from the Xian University of Electronic Science and Technology, Xian, China, in 2010, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2016. He used to be interested in research topics in computer networks, including router architecture, fast packet processing, and future Internet architecture, such as Named-Data Networking and Software-Defined Networking.

Yi Wang is now a Research Associate Professor in the SUSTech Insitute of Future Networks, Southern University of Science and Technology, Shenzhen, China, 518055. He received the Ph.D. degree in Computer Science and Technology from Tsinghua University in July, 2013. His research interests include router architecture design and implementation, software-defined networks, greening the Internet, fast packet forwarding, information-centric networking, and time-sensitive networks.

Gong Zhang is a Chief Architect Researcher Scientist, director of the Future Network Theory Lab. His major research directions are network architecture and large-scale distributed systems. He has abundant R&D experience on system architect in networks, distributed system and communication system for more than 20 years. He has more than 90 global patents in which some play significant roles in the company. In 2000, he acted as a system engineer for L3+ switch product and became the PDT (Product development team) leader for smart device development, pioneering a new consumer business for the company since 2002. Since 2005, he was a senior researcher, leading future internet research and cooperative communication. In 2009, he was in charge of the advance network technology research department, leading researches of future network, distributed computing, Database system and data analysis. In 2012, he became the Principal Researcher and led the system group in data mining and machine learning.

Bin Liu received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively. He is now a Full Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His current research areas include high-performance switches/routers, network processors, high-speed security, and greening the Internet. He has received numerous awards from China, including the Distinguished Young Scholar of China and won the inaugural Applied Network Research Prize sponsored by ISOC and IRTF in 2011.