Ultra-Fast Bloom Filters using SIMD Techniques

Jianyuan Lu[†], Ying Wan[†], Yang Li[†], Chuwen Zhang[†], Huichen Dai[†], Yi Wang[§], Gong Zhang[§] and Bin Liu[†]
[†] Tsinghua National Laboratory for Information Science and Technology
[†] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[§] Huawei Future Network Theory Lab

***Abstract*—The network link speed is increasing at an alarming rate, which requires all network functions on routers/switches to keep pace. Bloom filter is a widely-used membership check data structure in network applications. 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, by leveraging the SIMD techniques. We make three improvements for the 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 change a Bloom filter's bit-test process from sequential to parallel. Third, we increase the cache efficiency of membership check by encoding an element's information to a small block which can easily fit into a cache-line. Both theoretical analysis and extensive simulations show that the UFBF greatly exceeds the state-of-the-art Bloom filter variants on membership check speed.**

I. Introduction

Bloom filters are space-efficient randomized data structures for membership check [1]. Due to simplicity and efficiency, Bloom filters (and their variants) have been applied in a wide range of network applications [2, 3]. The trend for these network applications is that they will run in higher and higher speed network environment. The 40GE and 100GE ports for routers' line-cards have already been commercialized and deployed [4]. High-end core routers such as Cisco CRS-X [5] and Huawei NE9000 [6] 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 processing delay for network devices to process a packet. For example, a 40GE port needs to achieve 60Mpps throughput, *i.e.*, processing a packet within 75 clock cycles for a state-of-the-art 4GHz CPU. Therefore, Bloom filters need to run extremely fast in practice to avoid becoming the network applications' performance bottleneck.

Most of the network applications assume 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 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 [7]. Though simple hash functions can be the alternatives for Bloom filters [8], their computational cost cannot 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 may cause several cache misses. Due to 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 [9] or use a large number of hash functions in one Bloom filter [10], will experience more obvious computational cost and memory access delay.

In this paper, we propose a new Bloom filter variant called Ultra-Fast Bloom Filter (UFBF), which aims to improve a Bloom filter's membership check speed. The UFBF mainly uses the SIMD[1] techniques to accelerate the membership check. The UFBF consists of a sequence of blocks. An element's information is encoded in a randomly selected block. We make three optimizations for the UFBF to ensure it run fast. First, we develop a novel hash computation algorithm which can compute k hash functions in parallel. The hash functions in UFBF originate from the same hash function with different hash seeds. The parallel hash computation algorithm initializes an SIMD register to these hash seeds, and then operates them in parallel to get k hash values. The UFBF reduces hash computation time to (approximate) $1/k$ of standard Bloom filter. Second, UFBF changes the sequential bit-test process to parallel bit-test process. The standard Bloom filter has to test the membership bits one by one. However, the UFBF can test the membership 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 each word is associated with one hash function. That is to say, a hash function can only address its associated word. Third, the UFBF improves the cache efficiency by encoding an element's information to a 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 could occur during an element's

This work is sponsored by Huawei Innovation Research Program (HIRP), NSFC (61602271, 61373143, 61432009), China Postdoctoral Science Foundation (No. 2016M591182), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20130002110084). Corresponding Author: Bin Liu (liub@tsinghua.edu.cn)

[1]Single Instruction Multiple Data (SIMD) instructions can operate multiple operands in parallel. They are widely supported by general CPUs and embedded CPUs. Take Intel CPU as an example. It supports *MMX, SSE, AVX, FMA, KNC, SVML etc*, SIMD instruction sets [11].

978-1-5386-2704-4/17/$31.00 ©2017 IEEE

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 presents the experimental results for performance evaluation. Section V concludes the paper.

II. Related Work

Bloom filter was introduced by Burton H. Bloom in 1970 [1], which is called Standard Bloom Filter (SBF) in this paper. We assume the readers have the basic knowledge of this data structure. 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 [12]. 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. One-Memory Bloom Filter (OMBF) [13] 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. OMBF effectively reduces (at most) k cache misses to (at most) one cache miss in an element's membership check. Blocked Bloom Filter (BBF) [14] has a similar framework to 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.

2) Reducing hash computational cost. Lu *et al.* [15] 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. Kirsh and Mitzenmacher [16] 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 [17]. However, the paper lacks theoretical analysis on the independence of the additional synthetic hash functions.

III. Ultra-Fast Bloom Filters

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 $m = r * b = r * k * w$.

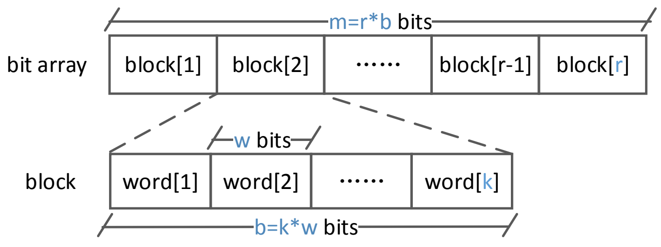


Fig. 1. The basic structure of UFBF.

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. An insertion example of UFBF is shown in Figure 2, where $w = 4$.

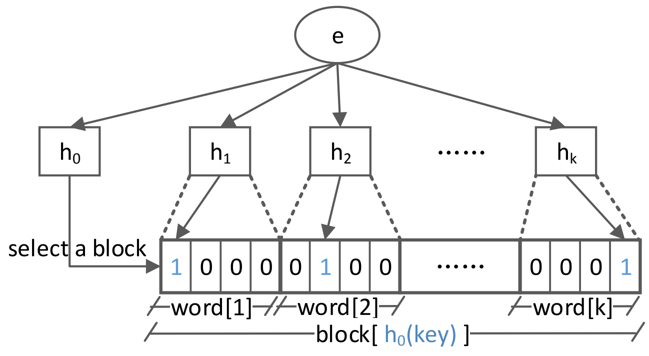


Fig. 2. An example of the insertion process in UFBF.

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 Parallel Hash Computation Algorithm

We develop a novel algorithm for UFBF to calculate k hash functions in parallel with the use of SIMD instructions. The hash computation algorithm in UFBF is designed to facilitate the use of SIMD instructions. 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.

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 set 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 using the same hash function with different initial seeds. It first loads p seeds 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. 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.

Algorithm 1: The parallel hash computation algorithm in UFBF

```
1 $seeds[p] \leftarrow [seed_1, seed_2, \ldots, seed_p]$
2 $hashVals[p] \leftarrow [0, 0, \ldots, 0]$
3 $vr\_seeds \leftarrow$ v_load($seeds$)
  /* load the $p$ seeds to an SIMD register   */
4 $vr\_val \leftarrow$ v_hashFunc($vr\_seeds$)
  /* implement the SIMD hash function which
     takes $p$ seeds and computes in parallel */
5 $hashVals \leftarrow$ 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 $val \leftarrow val$ OP $a$
  /* OP is a general arithmetic operation, $val$
     stores the intermediate hash value     */
        $\Downarrow$
1 $vr\_a \leftarrow$ v_broadcast($a$)
  /* v_broadcast copy $p$ copies of $a$ to $vr\_a$ */
2 $vr\_val \leftarrow$ v_OP($vr\_val, vr\_a$)
  /* v_OP is the SIMD-version of OP, $vr\_val$
     stores the intermediate $p$ hash values  */
```

C. Parallel Bit-Test in Membership Check

In the membership check process, standard Bloom filter (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 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 UFBF encodes an element to k consecutive words (in a block), and these words can be fetched and tested at the same time.

The membership check algorithm for UFBF is shown in Algorithm 3. In this algorithm, the k membership bits are tested in parallel. The k hash functions for membership check 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.

Algorithm 3: The membership check algorithm in UFBF

```
1  Function membershipCheck(element e)
2      $loc \leftarrow$ compute the block index of $e$
3      $vr\_val \leftarrow$ compute $k$ hash values using Algorithm 1
4      $vr\_a \leftarrow$ v_broadcast(1)
5      $vr\_a \leftarrow$ v_shiftLeft($vr\_a, vr\_val$)
       /* v_shiftLeft shifts $k$ words in $vr\_a$
          left in parallel by the amount
          specified by $vr\_val$                 */
6      $vr\_b \leftarrow$ v_load(&$block[loc]$)
7      $vr\_b \leftarrow$ v_not($vr\_b$)
       /* v_not is bitwise NOT operation        */
8      v_test($vr\_a, vr\_b$)
       /* v_test is bitwise AND operation       */
9      if zero-flag is set then
10         return positive
11     end
12     return negative
13 end
```

D. Cache Efficiency for Membership Check

The cache efficiency for membership check of UFBF is expected to far better than SBF. 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. 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 cache.

Formally, we analyze the worst case cache misses in one membership check of UFBF. In Theorem 1, we prove that only one cache miss would occur in worst case in one membership check process, if the block size divides the cache-line size and the bit array is cache-line size aligned. In practice, the size of cache-line is usually 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 $b|L$ 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 $addr\_s$, $addr\_e$. 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 and we can get $0 < (t-i)s + j < 1$. Apparently, $(t-i)s + j$ is an integer and we get a contradiction. $\square$

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 in a membership block, and each word selects one bit. 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:

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

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

$$Pr\{\mathcal{X} = x\} = \binom{n}{x}\left(\frac{1}{r}\right)^x\left(1 - \frac{1}{r}\right)^{n-x}, \forall\ 0 \leq x \leq n \tag{2}$$

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

$$\begin{aligned} f_u = Pr\{\mathcal{F}\} &= \sum_{x=0}^{n}(Pr\{\mathcal{X} = x\} \cdot Pr\{\mathcal{F}|\mathcal{X} = x\}) \\ &= \sum_{x=0}^{n}\binom{n}{x}\left(\frac{1}{r}\right)^x\left(1 - \frac{1}{r}\right)^{n-x}\left(1 - \left(1 - \frac{1}{w}\right)^x\right)^k \end{aligned} \tag{3}$$

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 I presents the theoretical false positive probability comparison between SBF (f_s) and UFBF (f_u). We 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. It can be concluded from this table that UFBF has higher false positive probability than SBF. However, UFBF has extremely faster membership check speed than SBF (shown in Section IV), which can compensate the higher false positive drawback.

TABLE I
THEORETICAL FALSE POSITIVE PROBABILITY COMPARISON 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

IV. EVALUATION

We make several experiments to evaluate our proposed UFBF using real-world Internet traces.

A. Experiment Setup

Platform: We implement the experiments on a commodity server with Intel CPU Core i7-4790 (4 cores $\times$ 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 mainly 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 [11]. *AVX* can implement 8 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 [18]. 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 [19], to evaluate the performance of UFBF. The trace is extracted from a backbone 10Gbps link and lasts 60 minutes. We use two datasets extracted from the traces for our evaluation, as shown in Table II.

TABLE II
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 *MurmurHash3* [20] and *lookup3* [21] 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 shows 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. We can conclude that the more hash functions used, the more time we can reduce for SIMD-version hash functions. The comparison of *MurmurHash3* and its SIMD-version has a similar result to *lookup3*.

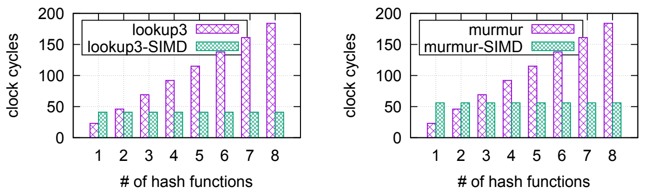


Fig. 3. (Left) The run time comparison between MurmurHash3 hash function and its SIMD-version; (Right) The run time comparison between lookup3 hash function and its SIMD-version.

C. Membership Check Speed Evaluation

We compare three Bloom filters, SBF [1], OMBF [13], and OHBF [15], 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 mainly attempts to reduce the hash computation overhead. 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.

Figure 4 presents the membership check speed comparison of 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. For UFBF, both positive check and negative check 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 5 presents the negative and positive membership check speed comparison of 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 L3-Cache.

D. False Positive Evaluation

We make two experiments on false positive ratio evaluation. Figure 6 presents the false positive ratio comparison between

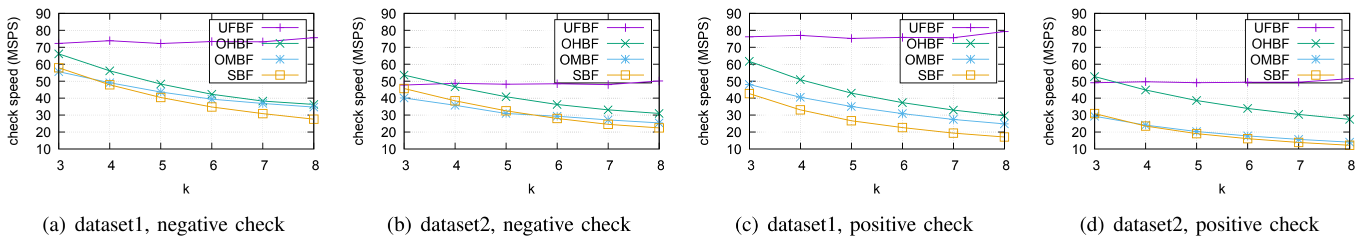
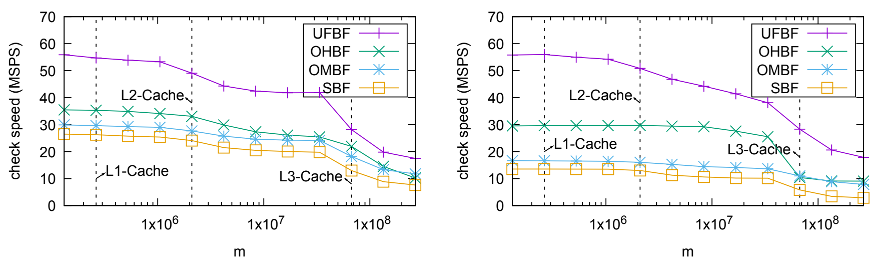

(a) dataset1, negative check (b) dataset2, negative check (c) dataset1, positive check (d) dataset2, positive check

Fig. 4. The membership check speed of the four Bloom filters: SBF, OMBF, OHBF, and UFBF. MSPS represents 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.



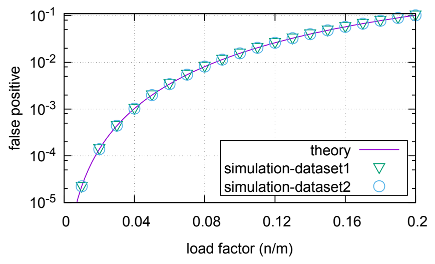


Fig. 5. The negative (Left) and positive (Right) membership check speed of the four Bloom filters: SBF, OMBF, OHBF, and UFBF. We set $k = 8$. The load factor is $\frac{n}{m} = 0.1$. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment.

Fig. 6. The false positive ratio of UFBF, $n = 10000, w = 4, k = 4$. Each point in this figure is the mean of 1,000 experiments.

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.

V. Conclusions

In this paper, we propose a new Bloom filter variant called Ultra-Fast Bloom Filter (UFBF), which speeds up the membership check process by the novel use of SIMD techniques. Since SIMD instructions are widely supported by most of the modern CPUs, our UFBF design has a very good application prospect. Different with traditional Bloom filters, the UFBF computes hash functions and implements bit-test process both in parallel. The UFBF also improves cache efficiency by encoding an element 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 dramatical improvement in performance, the tradeoff is absolutely worthwhile.

References

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

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

[3] S. 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.

[4] "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.

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

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

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

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

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

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

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

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

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