Internet Mathematics Vol. 7, No. 1: 28–44

The Power of $1 + \alpha$ for Memory-Efficient Bloom Filters

Evgeni Krimer and Mattan Erez

Abstract. This paper presents a cache-aware Bloom-filter algorithm with improved cache behavior and lower false-positive rates compared to prior work. The algorithm relies on the power-of-two choice principle to provide a better distribution of set elements in a blocked Bloom filter. Instead of choosing a single block, we insert new elements into the less-loaded of two blocks to achieve a low false-positive rate while performing only two memory accesses on each insert or query operation. The paper also discusses an optimization technique to balance cache effectiveness with the false-positive rate to fine-tune the Bloom-filter properties.

1. Introduction

A *Bloom filter* (BF) is a space-efficient data structure representing a set that provides add and probabilistic membership query operations with a certain rate of false positives, the false-positive rate, and no false negatives [Bloom 70]. Bloom filters are implemented in either hardware or software and are commonly used in various fields including networks [Broder and Mitzenmacher 04] and web services [Fan et al. 00]. The two main metrics of a Bloom filter are its performance (throughput and latency) and its false positive rate (FPR). Our work addresses both at the same time by offering the *power-of-*$(1 + \alpha)$ *blocked Bloom*

ISSN: 1542-7951 print

filter scheme, which introduces a parameterizable tradeoff between performance and FPR.

The performance of a BF often has a large impact on overall system performance. For example, BF query latency and bandwidth dictate the performance of WebCache servers, which implement a software BF [Fan et al. 00], and network routers, which use a hardware BF [Dharmapurikar et al. 03]. As shown in 2.1, BF query operations require k memory accesses to random locations, which stress the cache in software implementations and add cost and time to hardware BFs. By changing the algorithm to exploit *locality* (the likelihood that the system will reuse recently used data [Hennessy and Patterson 03]), the performance of BFs can be improved. For software implementations, increasing locality will decrease the number of external memory accesses due to a higher number of cache hits. For hardware BFs, locality enables increased lookup capacity by utilizing smaller independent blocks to construct the memory array [Dharmapurikar et al. 03]. Due to this duality of software and hardware implementations, throughout this paper we use the term *block* to represent a cache line, memory page, or memory bank, depending on the specific implementation.

The FPR of the BF also directly impacts overall system performance and behavior, because queries that are falsely answered as belonging to the set typically result in unnecessary costly work being performed. Thus, it is not enough to improve locality and performance, but the FPR must be controlled so that it remains within a reasonable bound of the classical BF algorithm.

This work explores a smooth tradeoff space between the performance and the FPR. Previous attempts at improving locality have so far been either too aggressive or too conservative and have not reached an overall optimal point of balancing performance and the FPR. In work motivated by a hardware implementation of a BF for a network router [Chen et al. 07], the authors suggest pairing hash functions such that even hash functions can choose any bit in the array but odd hash functions are restricted to choosing a bit in the same block as their corresponding even pair. Thus only $k/2$ blocks are accessed for each BF operation. Their simulation results show a minor effect on the FPR. On the other hand, the algorithm they present is still very conservative with respect to locality and still requires $\mathcal{O}(k)$ ($\frac{k}{2}$ to be exact) memory block accesses.

A different publication, from the perspective of a software BF implementation, presents the blocked Bloom filter [Putze et al. 07]. This extremely aggressive approach to exploiting locality requires only a single block access for every query operation. However, it significantly increases the FPR for some BF configurations, as we discuss in this paper.

Our approach offers a flexible and parameterizable BF algorithm for exploiting locality, and we summarize our main contributions below:

- We analyze the FPR of the blocked Bloom filter in detail and show how its worse FPR relative to the classical BF algorithm is a result of a poor distribution of elements between memory blocks.
- We present the *power-of-two blocked Bloom filter*, which improves the load balance of elements across memory blocks, thus minimizing the FPR increase relative to the classical BF algorithm for some configurations. We then derive and validate an analytical model for the FPR of the power-of-two blocked Bloom filter.
- We develop the *power-of-*$(1+\alpha)$ *blocked Bloom filter*, which enables a smooth tradeoff curve between the locality and the FPR by combining the blocked Bloom filter and the power-of-two blocked Bloom filter approaches. We also derive and validate an analytical model for the FPR of the power-of-$(1+\alpha)$ blocked Bloom filter.
- We show how to fine-tune the tradeoff options in general, and how to find the optimal mix of power-of-two and blocked Bloom filters for any given BF configuration.

The rest of the paper is organized as follows. In Section 2, we describe previously published algorithms of the (classical) Bloom filter and the blocked Bloom filter, discussing their performance and FPR metrics. In Section 3.1, we propose the power-of-two blocked Bloom filter along with its analytical model as an alternative to the blocked Bloom filter. Then, in Section 3.1, we present the power-of-$(1+\alpha)$ blocked Bloom filter, which allows a fine-grained tuning to optimize the performance vs. FPR tradeoff; we provide an analytical model for the power-of-$(1+\alpha)$ blocked Bloom filter algorithm. In Section 4, we compare the simulation results of all mentioned approaches in terms of the FPR and distribution load balance. We discuss the results and provide experimental evaluations of the optimal α parameter for different configurations of the BF. Finally, in Section 5, we summarize our findings and present plans for future work.

2. Background and Related Work

The Bloom filter is a relatively simple, yet efficient, probabilistic data structure that represents a set. In this section we begin with a brief overview of the algorithm and the FPR properties of the (classical) Bloom filter. Many published

Algorithm 1: Classical Bloom filter, k *block accesses on average.*

```
function Add(x)
    for i = 0 to k do
        mem(h_i(x)) ⇐ 1
    end for
end
```

```
function Query(x)
    for i = 0 to k do
        if mem(h_i(x)) == 0 then
            return NOT FOUND
        end if
    end for
    return FOUND
end
```

modifications to this classical BF exist in the literature, including one that uses the same power-of-two principle that is the basis of this paper [Lumetta and Mitzenmacher 07]. Most of this prior work, however, including [Lumetta and Mitzenmacher 07], does not address the performance (throughput/latency) of the BF. We limit the discussion in this paper to the aggressive blocked BF [Putze et al. 07] (Section 2.2), which significantly improves performance at the cost of a worse FPR. We omit the details of the very conservative approach to improved performance presented in [Chen et al. 07], since it reduces the number of accesses by only a factor of two, still requiring $\mathcal{O}(k)$ accesses.

2.1. The (Classical) Bloom Filter

This subsection provides a brief overview of the Bloom filter data structure, algorithm, and FPR derivation. The BF uses a vector of m bits, which represents the set of elements S (where $\|S\| = n$) and allows membership queries. The BF utilizes k hash functions $(h_1..h_k)$, with each function mapping a potential set element to a single bit location in the range of $[0, m)$.

As shown in Algorithm 1 (and using the symbols summarized in Table 1), adding an element is done by setting all the bits pointed to by the k hash functions for this element to 1. Note that some of these bits might already be set, because they were touched by prior add operations of other elements. To query membership of an element, all the bits pointed to by the results of the k hash functions are tested. Only if all of them are set is the query result positive. However, a false positive response is possible because all the relevant bits might have been set during the addition of unrelated elements. Thus a query result may be positive even when the element queried was never added to the BF. Using basic probability properties and algebra (refer to [Mitzenmacher and Upfal 05]

n	number of elements
c	bits per element
m	total memory size ($\equiv c \cdot n$) [bits]
B	block size [bits]
b	number of blocks ($\equiv m/B$)
k	number of hash functions
h_i	hash function selecting a bit out of the bit array : elem $\Rightarrow [0..m)$
hb_i	hash function selecting a bit out of a block : elem $\Rightarrow [0..B)$
g_i/g	hash function selecting a block : elem $\Rightarrow [0..b)$
$\mathrm{mem}(x)$	bit x of the bit array
$\mathrm{mem}[y](x)$	bit x of the block y

Table 1. Definition of symbols used.

for more detail), the FPR of the (classical) Bloom filter can be written as

$$\mathrm{FPR}_{\mathrm{basic}}(m,n,k) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k. \tag{2.1}$$

With respect to performance, the worst-behaving query operation would require accessing k different blocks, assuming that there are at least k blocks in the BF bit array. In practice, Lemma 6.1 proves that an average query would require accessing $k - \mathcal{O}(1/b)$ different blocks, and thus the average case quickly approaches the worst-case performance.

2.2. The Blocked Bloom Filter

The blocked Bloom filter attempts to improve query performance by exploiting *locality* [Putze et al. 07]. As described below, Algorithm 2 associates a single block of B bits with each element. In this way, each element is confined to a single block with different elements associated with different blocks, and potentially multiple elements associated with each block. As a result, it allows us to perform add/query operations with a single block access.

The FPR of the blocked Bloom filter is strongly dependent on load balancing among the blocks. Let $D_{\mathrm{blocked}}(j)$ be the probability of having exactly j elements

Algorithm 2: Blocked Bloom filter, *one block access.*

function Add(x)
  block $\Leftarrow g(x)$
  for $i = 0$ to k **do**
  mem[block]$(hb_i(x)) \Leftarrow 1$
  end for
end
	function Query(x)
	  block $\Leftarrow g(x)$
	  for $i = 0$ to k **do**
	  if mem[block]$(hb_i(x)) == 0$ **then**
	  return NOT FOUND
	  end if
	  end for
	  return FOUND
	end

in a block; then the FPR can be represented as [Putze et al. 07]

$$\begin{aligned} \mathrm{FPR}_{\mathrm{blocked}}(B,k) &= \sum_{j=0}^{n} D_{\mathrm{blocked}}(j) \cdot \mathrm{FPR}_{\mathrm{basic}}(B,j,k) \\ &= \sum_{j=0}^{\infty} D_{\mathrm{blocked}}(j) \cdot \mathrm{FPR}_{\mathrm{basic}}(B,j,k). \end{aligned} \tag{2.2}$$

An element has an equal probability, $1/b$ of being associated with any block. Therefore, the distribution of elements among the blocks can be modeled using a binomial distribution with parameter $p = 1/b$ for n elements: $D_{\mathrm{blocked}}(\,) = \mathrm{bionomial}(n, \frac{1}{b})$. Since n/b is a small constant, and $n/b \equiv B/c$, a Poisson approximation can be used [Putze et al. 07]:

$$D_{\mathrm{blocked}}(\,) = \mathrm{bionomial}\left(n, \frac{1}{b}\right) \approx \mathrm{Poison}\left(\frac{n}{b}\right) \equiv \mathrm{Poison}\left(\frac{B}{c}\right). \tag{2.3}$$

As proved by Lemma 6.2, however, in the special ideal case of a fully balanced distribution among the blocks, $\mathrm{FPR}_{\mathrm{blocked\_balanced}} = \mathrm{FPR}_{\mathrm{basic}}$. This observation, along with (2.2), leads us to conclude that improving the balance among the blocks will improve the FPR, whose lower bound will be $\mathrm{FPR}_{\mathrm{basic}}$ in case of a fully balanced distribution.

3. Proposed Algorithms

In this section, we present two novel BF modifications. Using the power-of-two principle, we improve the load balance among the blocks in order to reduce the FPR in accordance with our conclusion from Section 2.2 and introduce the

power-of-two blocked Bloom filter. Then, we present the power-of-$(1 + \alpha)$ blocked Bloom filter, which is a combination of the power-of-two blocked Bloom filter and the existing blocked BF that provides a smooth tradeoff between the FPR and performance. For both algorithms, we develop an analytical model of the FPR and validate them later in Section 4.

3.1. The Power-of-Two Blocked Bloom Filter

The distribution of elements among the blocks is similar to a problem of random distribution of n balls into b bins. The power-of-two idea described in the literature [Mitzenmacher and Upfal 05] suggests that instead of placing the ball into a randomly selected bin, we should select $d \geq 2$ bins and choose the least-loaded among them. This approach would decrease the number of the balls in the bin that contains the most from $\Theta\left(\frac{\ln n}{\ln(\ln n)}\right)$ to $\Theta\left(\ln\left(\ln n\right)\right)$.

We propose a modification to the blocked BF that uses two hash functions, g_1 and g_2, to select two blocks for each element. For the *add* operation, we choose the less-loaded block of the two, and the element is added to the chosen block in the same way as in the blocked BF. To perform a membership query, we have to check both blocks that are associated with the element, because we do not know to which of the two blocks the element was added. Thus, the result will be positive if either of the two blocks returns a positive result. In other words, only if the query result is negative for both blocks will the outcome be negative. Algorithm 3 presents the algorithm.

Making the algorithm access two blocks and selecting the less-loaded block improves the load balance between the blocks (see Figure 1). The FPR of the power-of-two blocked BF is given by

$$\mathrm{FPR}_{\mathrm{power-of-2}}(B,k) = 2 \cdot \sum_{j=0}^{\infty} D_{\mathrm{power-of-2}}(j) \cdot \mathrm{FPR}_{\mathrm{basic}}(B,j,k), \tag{3.1}$$

and it can be derived in a similar fashion to the FPR of the blocked BF, (2.2), with two important changes. First, the distribution $D_{\mathrm{power-of-2}}(x)$ is different from the distribution of the blocked BF $D_{\mathrm{blocked}}(x)$ and is more balanced. Second, unlike the blocked BF, querying the power-of-two blocked BF selects two blocks and queries each of them independently, and hence the FPR requires a coefficient of 2.

The distribution of elements among the blocks using the power-of-two blocked BF $D_{\mathrm{power-of-2}}(x)$ differs from that generated by the blocked BF $D_{\mathrm{blocked}}(x)$ as depicted in Figure 1. Therefore, $\mathrm{FPR}_{\mathrm{power-of-2}}(B,k) \neq 2\mathrm{FPR}_{\mathrm{blocked}}(B,k)$.

Algorithm 3: Power-of-two blocked Bloom filter, *two block accesses.*

```
function Add(x)
    block ⇐ choose less loaded
            block (g1(x), g2(x))
    for i = 0 to k do
        mem[block](hbi(x)) ⇐ 1
    end for
end
```

```
function Query(x)
    block1 ⇐ g1(x)
    block2 ⇐ g2(x)
    for i = 0 to k do
        if mem[block1](hbi(x)) == 0 then
            for j = 0 to k do
                if mem[block2](hbj(x)) == 0 then
                    return NOT FOUND
                end if
            end for
            return FOUND
        end if
    end for
    return FOUND
end
```

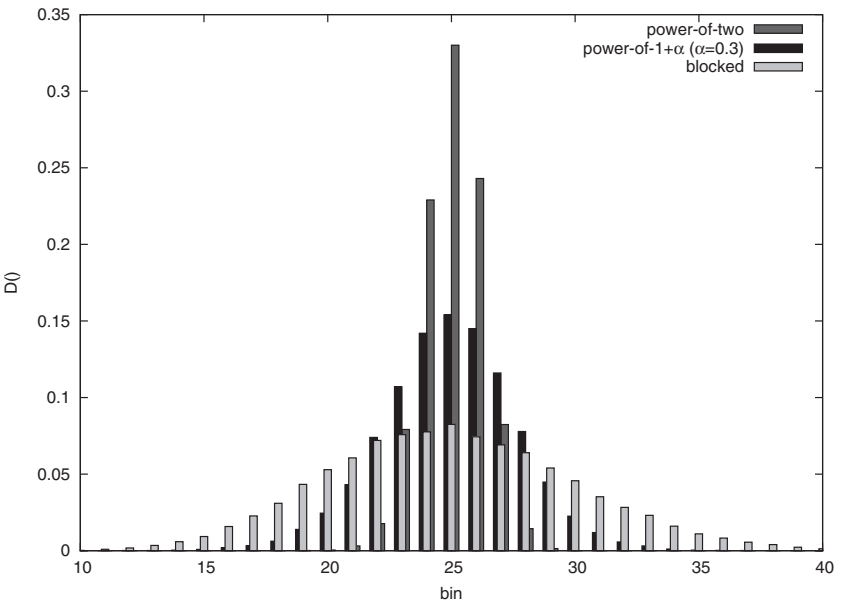

Figure 1. Distribution $D(\,)$ of various approaches after inserting $n = 10^6$ elements with $c = 20$ bits per element.

Moreover, as will be discussed and proved in Section 4, $\mathrm{FPR}_{\mathrm{power-of-2}}$ is actually lower than $\mathrm{FPR}_{\mathrm{blocked}}$ for a range of configurations.

The distribution $D_{\mathrm{power-of-2}}(x)$ is derived iteratively by analyzing the process of adding elements to the power-of-two blocked BF structure. After adding an element, there are three possible outcomes: the number of blocks having x elements can increase by 1 if the element was added to a block with $x-1$ elements (so that it becomes a new block with x elements); it can decrease by 1 if an element was added to a block with x elements (which becomes a block with $x+1$ elements), or finally, it can remain unchanged.

The algorithm selects two blocks and chooses the less-loaded one. In order to choose the block with exactly x elements, either both selected blocks should contain x elements or one block should contain x elements and the second block any greater number. The order of selecting the blocks is unimportant. Let $P_{\mathrm{choose}}(x)$ be the probability of choosing to add the new element to a block with exactly x elements. Then, we can derive $P_{\mathrm{inc}}(x)$ as the probability of increasing the number of blocks with exactly x elements and $P_{\mathrm{dec}}(x)$ as the probability of decreasing the number of blocks with exactly x:

$$\begin{aligned} P_{\mathrm{choose}}(x) &= \left(D_{\mathrm{power-of-2}}(x)\right)^2 + 2 \cdot D_{\mathrm{power-of-2}}(x) \cdot \sum_{j=x+1}^{\infty} D_{\mathrm{power-of-2}}(j), \\ P_{\mathrm{inc}}(x) &= P_{\mathrm{choose}}(x-1), \\ P_{\mathrm{dec}}(x) &= P_{\mathrm{choose}}(x). \end{aligned} \tag{3.2}$$

Let $N_n(x)$ be the number of blocks with x elements after the nth element insertion, we can derive $D_{\mathrm{power-of-2}}(x)$:

$$\begin{aligned} D_{\mathrm{power-of-2}}(x) &= \frac{N(x)}{b}, \\ N_{n+1}(x) &= N_n(x) + 1 \cdot P_{\mathrm{inc}}(x) - 1 \cdot P_{\mathrm{dec}}(x), \\ \frac{\partial N(x)}{\partial n} &= P_{\mathrm{inc}}(x) - P_{\mathrm{dec}}(x). \end{aligned}$$

The derivation of $D_{\mathrm{power-of-2}}(x)$ is given by

$$\frac{\partial D_{\mathrm{power-of-2}}(x)}{\partial n} = \frac{1}{b} \cdot \frac{\partial N(x)}{\partial n} = \frac{1}{b} \cdot \left(P_{\mathrm{choose}}(x-1) - P_{\mathrm{choose}}(x)\right). \tag{3.3}$$

We solve it numerically to obtain $\mathrm{FPR}_{\mathrm{power-of-2}}(B,k)$ using (3.1). These results are validated and compared with other algorithms in Section 4.

Algorithm 4: Power-of-$1 + \alpha$ Bloom filter, $1 + \alpha$ *block accesses on average.*

function Add(x)	function Query(x)
if $\Phi(x)$ **then**	**if** $\Phi(x)$ **then**
$\text{Add}_{\text{power-of-2}}(x)$	return $\text{Query}_{\text{power-of-2}}(x)$
else	**else**
$\text{Add}_{\text{blocked}}(x)$	return $\text{Query}_{\text{blocked}}(x)$
end if	**end if**
end	end

3.2. The Power-of-$(1 + \alpha)$ Blocked Bloom Filter

In this subsection, we introduce the power-of-$(1 + \alpha)$ blocked Bloom filter. This approach allows fine-grained control over the distribution balance, the FPR, and the number of blocks to be accessed by combining the blocked BF with the power-of-two blocked BF.

The tradeoff is controlled using the $\Phi(x)$ function, which uses $0 \leq \alpha \leq 1$, defined as follows:

$$\Phi(x) = \begin{cases} 0 & \text{with probability } 1 - \alpha, \\ 1 & \text{with probability } \alpha. \end{cases}$$

Note that although $\Phi(x)$ is a probabilistic function, it is deterministic, i.e., a result for a given element never changes. Algorithm 4 shows the use of $\Phi(x)$ in the *add* and *query* operations.

Deriving $\text{FPR}_\alpha(B, k)$ is similar to the derivation of $\text{FPR}_{\text{power-of-2}}(B, k)$ (see (3.1)), although the coefficient will be $1 + \alpha$ rather than 2 and the distribution is different as well, as shown in the following equation:

$$\begin{aligned} \text{FPR}_\alpha(B, k) &= \alpha \cdot \left[2 \cdot \sum_{j=0}^{\infty} D_\alpha(j) \cdot \text{FPR}_{\text{basic}}(B, j, k) \right] \\ &\quad + (1 - \alpha) \cdot \left[\sum_{j=0}^{\infty} D_\alpha(j) \cdot \text{FPR}_{\text{basic}}(B, j, k) \right] \\ &= (1 + \alpha) \cdot \sum_{j=0}^{\infty} D_\alpha(j) \cdot \text{FPR}_{\text{basic}}(B, j, k). \end{aligned} \tag{3.4}$$

The probability of using the power-of-two blocked BF algorithm with a coefficient of two is α. On the other hand, the probability of using the blocked BF algorithm with a coefficient of one is $1-\alpha$. One can observe that for $\alpha = 0$, this scheme reduces to the blocked BF, whereas for $\alpha = 1$, it turns into the power-of-two blocked BF.

One can calculate $D_\alpha(x)$ iteratively. Adding an element to the data structure with probability α will follow the power-of-two blocked BF algorithm and update the distribution in a way similar to (3.3). As before, with probability $1-\alpha$, the add operation will follow the blocked BF algorithm and update the distribution according to (6.2). Lemma 6.3 proves that the distribution defined by (2.3) is the solution of the iterative (6.2). To simplify the equation, we define $P_{\alpha\_\text{choose}}(x)$ similarly to $P_{\text{choose}}(x)$ in (3.2) and use it in the following:

$$\begin{aligned} P_{\alpha\_\text{choose}}(x) &= (D_\alpha(x))^2 + 2 \cdot D_\alpha(x) \cdot \sum_{j=x+1}^{\infty} D_\alpha(j), \\ \frac{\partial D_\alpha}{\partial n} &= \alpha \cdot \left[\frac{1}{b} \cdot (P_{\alpha\_\text{choose}}(x-1) - P_{\alpha\_\text{choose}}(x))\right] \\ &\quad + (1-\alpha) \cdot \left[\frac{1}{b}(D_\alpha(x-1) - D_\alpha(x))\right]. \end{aligned} \tag{3.5}$$

We solve (3.5) numerically to obtain $\text{FPR}_\alpha(B,k)$ using (3.4). These results are validated and compared with other algorithms in Section 4.

4. Results and Discussion

This section compares the FPR and distribution metrics of the schemes described in our work. Results are generated using a simulator written in C++ and run on an Intel Core CPU. We use the linear congruential algorithm with 48-bit integer arithmetic (drand48) to generate pseudorandom numbers to be used as the hash functions for adding elements. Each simulation is repeated 100 times, and the FPR is derived from the number of set bits after 10^6 random elements are added. Table 2 summarizes the parameters used in the simulations.

In Section 2.2, we mentioned that improving balance among blocks improves the FPR. The power-of-two blocked BF uses the power-of-two principle to achieve this. In the power-of-$(1+\alpha)$ blocked BF approach we introduce α to control the balance. Figure 1 shows experimental results for the distributions $D(x)$ of all three approaches. The benefit of the power-of-two principle is clearly seen when the $1+\alpha$ approach (with $\alpha = 0.3$) is, as expected, between the blocked and power-of-two blocked BFs.

n	number of elements	10^6
c	bits per element	(1..40)
m	total memory size	$\equiv c \cdot n$ [bits]
B	block size	500 [bits]
b	number of blocks	$\equiv m/B$
k	number of hash functions	$\lvert \ln 2 \cdot c \rvert_{\text{round}}$

Table 2. Parameters used for simulation.

The FPRs of the different approaches are depicted in Figure 2. It clearly shows the disadvantage of the blocked BF against the power-of-two blocked BF for configurations with $c \geq 17$. The figure also shows how the power-of-$(1 + \alpha)$ blocked BF ($\alpha = 0.3$) enables greater flexibility and yields the best overall FPR for $13 \leq c \leq 20$. Finally, Figure 2, validates the analytical models and shows a good match with the simulation results for all three schemes.

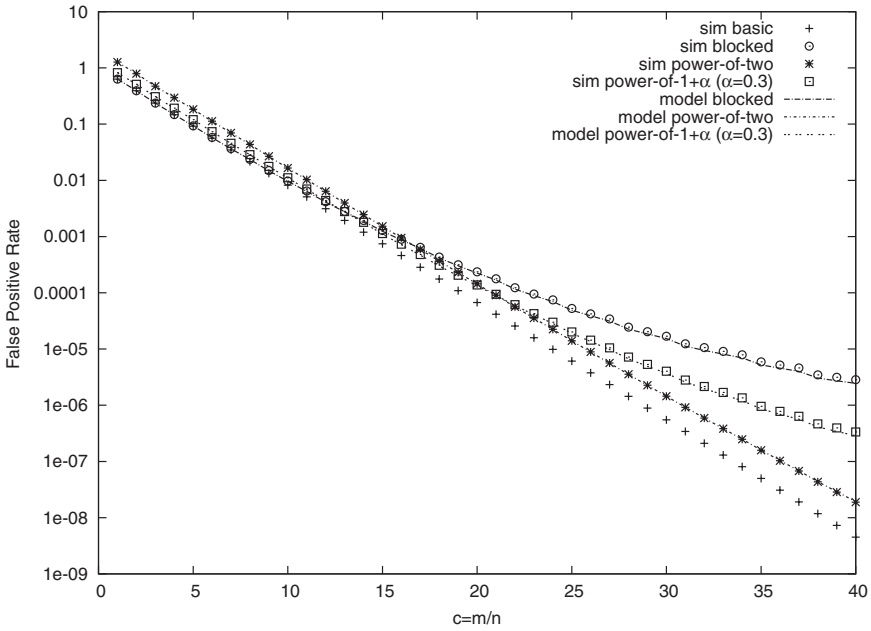

Figure 2. False positive rates of different Bloom filter approaches for varying values of c (bits per element).

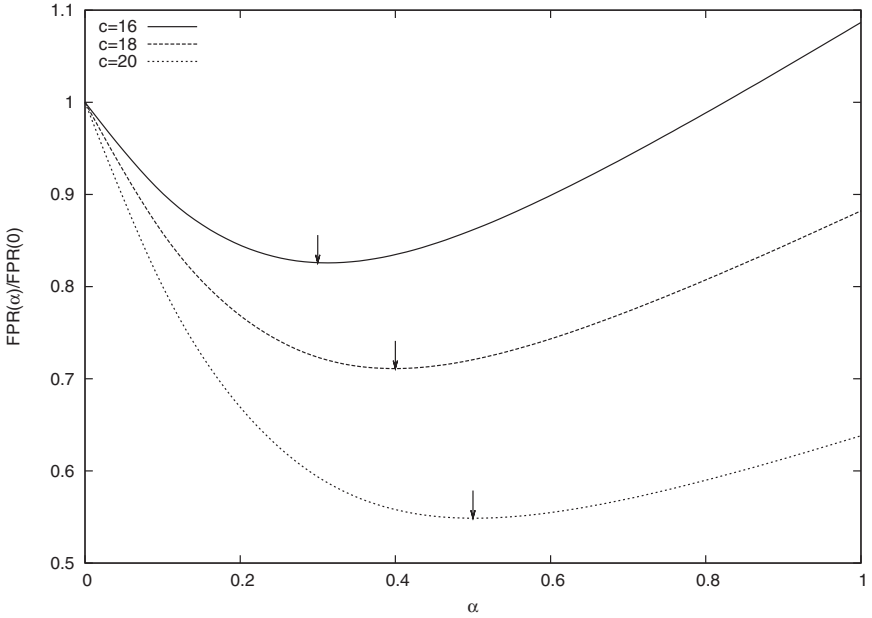

Figure 3. Values of FPR(α)/FPR(0) for different values of c $(16, 18, 20)$.

The parameter α in the power-of-$(1+\alpha)$ blocked BF controls the algorithm. As mentioned before, both the blocked BF and the power-of-two blocked BF are special cases of the power-of-$(1+\alpha)$ blocked BF for $\alpha = 0$ and $\alpha = 1$ respectively. Therefore, to choose the best scheme for each BF parameter configuration, an optimal α value should be selected.

Figure 3 shows the ratio of the power-of-$(1+\alpha)$ blocked BF FPR versus the blocked BF FPR as a function of α for different configurations of c. This function has a convex form where an optimal α exists. As an example, Figure 3 shows the optimal points for $c = 16, 18, 20$, which are respectively $\alpha = 0.3, 0.4, 0.5$.

Figure 4 plots the optimal α values across a range of BF configurations (varying c). The figure clearly shows that for $c \leq 10$, the best FPR is achieved with $\alpha = 0$, i.e., the blocked BF; the power-of-two blocked BF is optimal for $c \geq 31$; and the optimal α in between can be approximated linearly.

5. Summary and Future Work

The Bloom filter is an efficient and widely used probabilistic data structure implemented both in software and hardware. In Section 2.1, we summarized the

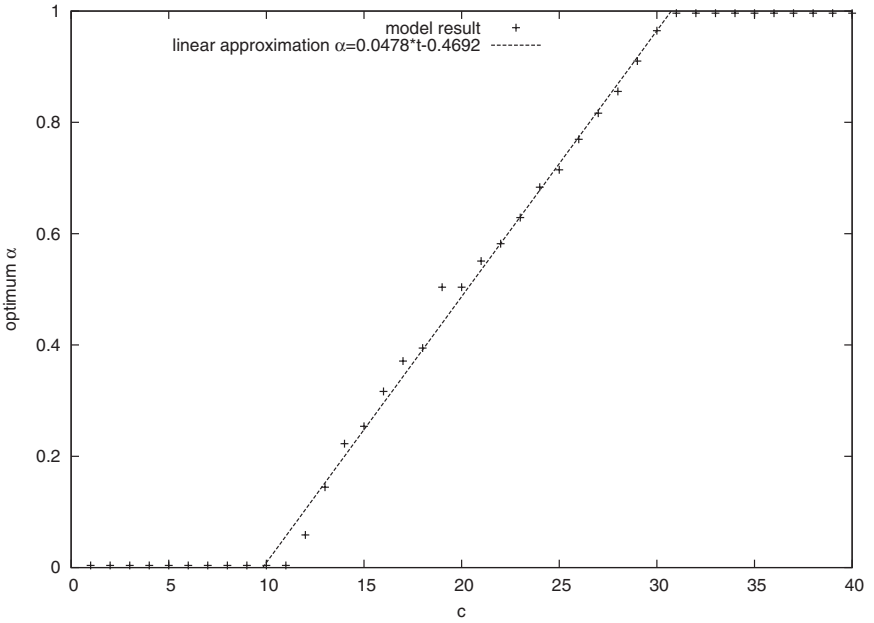

Figure 4. Optimal α for different values of c can be linearly approximated.

(classical) BF algorithm, derived its FPR, and showed how its performance is disadvantaged due to accesses to multiple (k) memory blocks. We then described the blocked BF [Putze et al. 07], which is a high-performance alternative, which requires only a single memory block access. The blocked BF, however, introduces a higher false positive rate (FPR). We discussed the reason for the FPR increase and showed that by balancing the load of elements among the blocks, the FPR could be improved.

We followed this conclusion and developed the power-of-two blocked BF, which improves the balance of element distribution using the power-of-two choice principle (Section 3.1), and derived an analytical model for its FPR. Our analysis showed, however, that for some configurations, the blocked BF still has a lower FPR compared to the power-of-two blocked BF; therefore, we introduced the power-of-$(1 + \alpha)$ blocked BF to blend the advantages of both the blocked and power-of-two blocked schemes (Section 3.2). Using the parameter α, the power-of-$(1 + \alpha)$ blocked BF approach allows a fine-grained tuning of the algorithm for an optimal FPR.

Using the analytical model we presented for the power-of-$(1 + \alpha)$ blocked BF, we analyzed the FPR's dependence on α and depicted it in Figure 4. We found

that it can be approximated using a linear function; however, we leave deeper analysis of this dependency to future work.

6. Appendix

Lemma 6.1. *Let k be the number of hash functions used in a (classical) BF and let b be the number of blocks that the memory array comprises. Then the average query operation will require accesses to $k - \mathcal{O}(1/b)$ different blocks.*

Proof. Each query operation requires accesses to k different bits in the memory array. For a single bit lookup, the probability that a specific block is accessed is $1/b$. Conversely, the probability that a block is not accessed is $1 - 1/b$. Therefore, the probability that a block is not accessed for any lookups of the k bits is $(1 - 1/b)^k$, and the probability that the block is accessed by at least a single bit lookup is $1 - (1 - 1/b)^k$. Based on this, the average number of blocks accessed by a query operation is $b\left[1 - (1 - 1/b)^k\right]$. Using the binomial expansion, we derive the required average number of accessed blocks:

$$b \cdot \left[1 - \left(1 - \frac{1}{b}\right)^k\right] = b \cdot \left[1 - \left(1 - \frac{k}{b} + \mathcal{O}\left(\frac{1}{b^2}\right)\right)\right] = k - \mathcal{O}\left(\frac{1}{b}\right).$$

□

Lemma 6.2. *In the special case of a fully balanced distribution among the blocks of a blocked BF containing a total of n elements in m bits of memory split into b blocks, let $D_{\text{blocked\_balanced}}(j)$ be the probability of having exactly j elements in a block. Then $\text{FPR}_{\text{blocked\_balanced}}$ is equal to $\text{FPR}_{\text{basic}}$.*

Proof. Since the blocks are balanced, the number of elements in each block is equal to n/b; therefore, $D_{\text{blocked\_balanced}}(j)$ is defined as

$$D_{\text{blocked\_balanced}}(j) = \begin{cases} 0, & j \neq \frac{n}{b}, \\ 1, & j = \frac{n}{b}. \end{cases} \tag{6.1}$$

Utilizing (6.1) in (2.2) yields

$$\begin{aligned} \text{FPR}_{\text{Blocked\_Balanced}}(B, k) &= \sum_{j=0}^{\infty} D_{\text{blocked\_balanced}}(j) \cdot \text{FPR}_{\text{basic}}(B, j, k) \\ &= \text{FPR}_{\text{basic}}\left(B, \frac{n}{b}, k\right) = \text{FPR}_{\text{basic}}(m, n, k). \end{aligned}$$

□

Lemma 6.3. *Let $D_{\text{blocked}}(j)$ be the probability of having exactly j elements in a block of a blocked BF as defined by (2.3). Then $D_{\text{blocked}}(j)$ solves the following iterative equation:*

$$\frac{\partial D_{\text{blocked}}(x)}{\partial n} = \frac{1}{b}\left(D_{\text{blocked}}(x-1) - D_{\text{blocked}}(x)\right). \tag{6.2}$$

Proof. We prove the result using (2.3) in (6.2). According to 2.3,

$$\begin{aligned}
D_{\text{blocked}}(x) &\approx \text{Poison}\left(\frac{n}{b}\right) = \frac{\left(\frac{n}{b}\right)^x \cdot e^{-\frac{n}{b}}}{x!}, \\
\frac{\partial D_{\text{blocked}}(x)}{\partial n} &= \frac{\partial}{\partial n}\left(\frac{\left(\frac{n}{b}\right)^x \cdot e^{-\frac{n}{b}}}{x!}\right) = \frac{\frac{x \cdot n^{x-1}}{b^x} \cdot e^{-\frac{n}{b}}}{x!} \\
&\quad + \frac{\left(\frac{n}{b}\right)^x \cdot e^{-\frac{n}{b}}}{x!} \cdot \left(-\frac{1}{b}\right) \\
&= \frac{1}{b} \cdot \left(\frac{\left(\frac{n}{b}\right)^{x-1} \cdot e^{-\frac{n}{b}}}{(x-1)!} - \frac{\left(\frac{n}{b}\right)^x \cdot e^{-\frac{n}{b}}}{x!}\right) \\
&= \frac{1}{b}\left(D_{\text{blocked}}(x-1) - D_{\text{blocked}}(x)\right).
\end{aligned}$$

Another way to prove the lemma is by constructing a similar derivation to the iterative method used in (3.3). □

Acknowledgments The authors would like to thank the anonymous reviewers, whose comments helped us to improve the paper significantly. We would also like to thank Isaac Keslassy for fruitful discussions at the early stages of this research and Mehmet Basoglu for his help with preparing this manuscript for publication. Finally, we thank Intel corporation for providing equipment and funds in support of this research.

References

[Bloom 70] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors." *Communications of the ACM* 13 (1970), 422–426.

[Broder and Mitzenmacher 04] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey." *Internet Mathematics* 1 (2004), 485–509.

[Chen et al. 07] Y. Chen, A. Kumar, and J. J. Xu. "A New Design of Bloom Filter for Packet Inspection Speedup." In *IEEE Global Telecommunications Conference, 2007. GLOBECOM'07*, pp. 1–5, 2007.

[Dharmapurikar et al. 03] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. "Deep Packet Inspection Using Parallel Bloom Filters." In *Proceedings of the 11th Symposium on High Performance Interconnects, 2003*, pp. 44–51, 2003.

[Fan et al. 00] 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)* 8 (2000), 281–293.

[Hennessy and Patterson 03] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Burlington, MA: Morgan Kaufmann, 2003.

[Lumetta and Mitzenmacher 07] S. Lumetta and M. Mitzenmacher. "Using the Power of Two Choices to Improve Bloom Filters." *Internet Mathematics* 4 (2007), 17–33.

[Mitzenmacher and Upfal 05] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge UK: Cambridge University Press, 2005.

[Putze et al. 07] F. Putze, P. Sanders, and J. Singler. *Cache-, Hash- and Space-Efficient Bloom Filters*, Lecture Notes in Computer Science 4525. New York: Springer, 2007.

Evgeni Krimer, Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX 78712 (krimer@mail.utexas.edu)

Mattan Erez, Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX 78712 (mattan.erez@mail.utexas.edu)

Received February 2, 2010; accepted July 4, 2010.