An Improved Construction for Counting Bloom Filters

Flavio Bonomi[1], Michael Mitzenmacher[2,*], Rina Panigrahy[3,**], Sushil Singh[1], and George Varghese[1,***]

[1] Cisco Systems Inc.
{bonomi, sushilks, gevarghe}@cisco.com
[2] Harvard University
michaelm@eecs.harvard.edu
[3] Stanford University
rinap@cs.stanford.edu

Abstract. A counting Bloom filter (CBF) generalizes a Bloom filter data structure so as to allow membership queries on a set that can be changing dynamically via insertions and deletions. As with a Bloom filter, a CBF obtains space savings by allowing false positives. We provide a simple hashing-based alternative based on d-left hashing called a d-left CBF (dlCBF). The dlCBF offers the same functionality as a CBF, but uses less space, generally saving a factor of two or more. We describe the construction of dlCBFs, provide an analysis, and demonstrate their effectiveness experimentally.

1 Introduction

A Bloom filter is an inexact representation of a set that allows for false positives when queried; that is, it can sometimes say that an element is in the set when it is not. In return, a Bloom filter offers very compact storage: less than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set. There has recently been a surge in the popularity of Bloom filters and variants, especially in networking [6]. One variant, a counting Bloom filter [10], allows the set to change dynamically via insertions and deletions of elements. Counting Bloom filters have been explicitly used in several papers, including for example [7, 8, 9, 10, 12, 18, 19].

In this paper, we present a new construction with the same functionality as the counting Bloom filter, based on d-left hashing. We call the resulting structure a d-left counting Bloom filter, or dlCBF. For the same fraction of false positives, the dlCBF generally offers a factor of two or more savings in space over the standard solution, depending on the parameters. Moreover, the construction is very simple and practical, much like the original Bloom filter construction. As counting Bloom filters are often used in settings where space and computation

* Supported in part by NSF grant CCR-0121154 and a research grant from Cisco.
** Part of this work was done while working at Cisco Systems, Inc.
*** Now back at U.C. San Diego.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 684–695, 2006.

are limited, including for example routers, we expect that this construction will prove quite useful in practice.

2 Background

2.1 Bloom Filters and Counting Bloom Filters

We briefly review Bloom filters; for further details, see [6]. A Bloom filter represents a set S of m elements from a universe U using an array of n bits, denoted by $B[1], \ldots, B[n]$, initially all set to 0. The filter uses a group H of k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, n\}$ that independently map each element in the universe to a random number uniformly over the range. (This optimistic assumption is standard and convenient for Bloom filter analyses.) For each element $x \in S$, the bits $B[h_i(x)]$ are set to 1 for $1 \leq i \leq k$. (A bit can be set to 1 multiple times.) To answer a query of the form "Is $y \in S$?", we check whether all $h_i(y)$ are set to 1. If not, y is not a member of S, by the construction. If all $h_i(y)$ are set to 1, it is assumed that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set is easily derived. If p is the fraction of ones in the filter, it is simply p^k. A standard combinatorial argument gives that p is concentrated around its expectation

$$\left(1 - (1 - 1/n)^{mk}\right) \approx \left(1 - \mathrm{e}^{-km/n}\right).$$

These expressions are minimized when $k = \ln 2 \cdot (n/m)$, giving a false positive probability f of $f \approx (1/2)^k \approx (0.6185)^{n/m}$. In practice, k must be an integer, and both n/m (the number of bits per set element) and k should be thought of as constants. For example, when $n/m = 10$ and $k = 7$ the false positive probability is just over 0.008.

Deleting elements from a Bloom filter cannot be done simply by changing ones back to zeros, as a single bit may correspond to multiple elements. To allow for deletions, a *counting Bloom filter* (CBF) uses an array of n counters instead of bits; the counters track the number of elements currently hashed to that location [10]. Deletions can now be safely done by decrementing the relevant counters. A standard Bloom filter can be derived from a counting Bloom filter by setting all non-zero counts to 1. Counters must be chosen large enough to avoid overflow; for most applications, four bits suffice [5, 10]. We generally use the rule of four bits per counter when comparing results of our data structure with a standard CBF, although we do note that this could be reduced somewhat with some additional complexity.

2.2 Related Work on Counting Bloom Filters

The obvious disadvantage of counting Bloom filters is that they appear quite wasteful of space. Using counters of four bits blows up the required space by a factor of four over a standard Bloom filter, even though most entries are zero.

Some work has been done to improve on this. The spectral Bloom filter was desigend for multi-sets, but also considers schemes to improve the efficiency of storing counters [7]. A paper on "optimal" Bloom filter replacements is another work in this vein [17], introducing a data structure with the same functionality as a counting Bloom filter that is, at least asymptotically, more space-efficient.

While this problem has received some attention, the previous work does not appear to give useful solutions. Spectral Bloom filters are primarily designed for multi-sets and skewed streams; we do not know of experiments or other evidence suggesting they are appropriate as a replacement for a CBF. The alternatives suggested in [17] do not appear to have been subject to experimental evaluation. Moreover, the schemes suggested in both of these papers appear substantially more complex than the standard counting Bloom filter scheme. This simplicity is not just useful in terms of ease of programming; for implementations in hardware, the simplicity of the Bloom filter scheme translates into very straightforward and clean hardware designs.

Our goal is to provide a scheme that maintains the simplicity of the original counting Bloom filter construction, and further is backed by experimental results demonstrating that the scheme is likely to be very useful in practice. We believe our work is novel in these regards. Our motivation for our general approach came about when considering generalizations of Bloom filters for state machines. See [4] for more details.

2.3 Background: *d*-Left Hashing

Our approach makes use of d-left hashing, a variation of the balanced allocations paradigm [1] due to Vöcking [20], which we now recall. Often this setting is described in terms of balls and bins; here, for consistency, we use the terms elements and buckets. We have a hash table consisting of n buckets. Initially they are divided into d disjoint subtables of n/d buckets. (For convenience we assume n/d is an integer.) We think of the subtables as running consecutively from left to right. Each incoming element is hashed to give it a collection of d possible buckets where it can be placed, one in each subtable. We assume in the analysis that these choices are uniform and independent. Each incoming element is placed in the bucket containing the smallest number of elements; in case of a tie, the element is placed in the bucket of the leftmost subtable with the smallest number of elements. To search for an element in the hash table, the contents of d buckets must be checked. Note that, if the bucket size is fixed a priori, there is the possibility of overflow. Various combinatorial bounds on the resulting maximum load have been proven [2, 20].

For our purposes, we are more interested in obtaining precise estimates of d-left hashing under constant average load per bucket. For the case where elements are only inserted, this can be obtained by considering the fluid limit, corresponding to the limiting case where the number of elements and buckets grow to infinity but with the ratio between them remaining fixed. The fluid limit is easily represented by a family of differential equations, as described in [5, 14]. The advantage of using the fluid limits in conjunction with simulation is that

it provides insight into how d-left hashing scales and the probability of overflow when fixed bucket sizes are used. Because of lack of space, we do not review the derivation of the differential equations. (See the full version for more details.)

Analyzing the behavior with deletions is somewhat more problematic in this framework, as one requires a suitable model of deletions. Good insight can be gained by the following approach. Suppose that we begin by inserting m elements, and then repeatedly, at each time step, delete an element chosen uniformly at random and then insert a new element. The corresponding fluid limit equations can be easily derived and are very similar to the insertion-only case. We can run the family of equations until the system appears to reach a steady state distribution. (Again, more details are in the full version.)

3 The d-Left CBF Construction

3.1 The Framework

Our goal is to design a structure that allows membership queries on a set S over a universe U that can change dynamically via insertions and deletions, although there will be an upper bound of m on the size of the set. A query on $x \in S$ should always return that $x \in S$; a query on some $y \notin S$ could give a false positive. The target false positive rate is ϵ. We allow for data structures that may with very small probability reach a failure condition, such as the overflow of a counter, at some point in its lifetime. Preferably, the failure probability is so small that it should not occur in practice. The failure condition should, however, be apparent, so that an appropriate response can be taken if necessary.

A standard counting Bloom filter offers one possible solution to this problem. Our alternative has a different starting point. It is a folklore result (see [6]) that if the set S is static, one can achieve essentially optimal performance by using a perfect hash function and fingerprints. One finds a perfect hash function $P : U \to [|S|]$, and then stores at each location an $f = \lceil \log 1/\epsilon \rceil$ bit fingerprint in an array of size $|S|$, computed according to some (pseudo-)random hash function H. A query on z requires computing $P(z)$ and $H(z)$, and checking whether the fingerprint stored at $P(z)$ matches $H(z)$. When $z \in S$ a correct response is given, and when $z \notin S$ a false positive occurs with probability at most ϵ; this uses $m\lceil \log 1/\epsilon \rceil$ bits.

The problem with this approach is that it does not cope with changes in the set S, and perfect hash functions are generally too expensive to compute for most applications. To deal with this, we make use of the fact, recognized in [5], that using d-left hashing provides a natural way to obtain an "almost perfect" hash function. The resulting hash function is only almost perfect in that instead of having one set element in each bucket, there can be several, and space is not perfectly utilized. A strong advantage, however, is that it can easily handle dynamically changing sets. The resulting construction meets our goals of being a substantial improvement over Bloom filters while maintaining simplicity. (See [12] for an alternative approach for dynamic "almost perfect" hash functions.)

3.2 The Construction of a *d*-Left Counting Bloom Filter

We first present a seemingly natural construction of a dlCBF that has a subtle flaw; we then demonstrate how this flaw can be corrected. To begin, we use a d-left hash table, where each bucket consists of many cells, each cell being a fixed number of bits meant to hold a fingerprint and a counter. As we want to avoid pointers to keep our representation as small as possible, we use a fixed number of cells per bucket, so that our hash table may be viewed as a large bit array.

We store a fingerprint for each element. The fingerprints are essentially compressed by taking advantage of how they are stored. Specifically, each fingerprint will consist of two parts. The first part corresponds to the *bucket index* the element is placed in. We assume the bucket index has range $[B]$, where in this setting we use $[x] = \{0, 1, \ldots, x-1\}$. The second part is the remaining fingerprint, which we refer to as the *remainder*, and is stored explicitly. We assume the remainder has range $[R]$.

For example, if we were just using a single hash function, and a single hash table with B buckets, we would use a hash function $H : U \to [B] \times [R]$. The m elements of S would be stored by computing $H(x)$ for each $x \in S$ and storing the appropriate remainders in a cell for each bucket. In order to handle deletions in the case that two (or more) elements might yield the same bucket and remainder, each cell would also contain a small counter. A false positive would occur if and only if for a query $y \notin S$ there existed $x \in S$ with $H(x) = H(y)$.

Using a single hash function yields the problem that the distribution of the load varies dramatically across buckets, essentially according to a Poisson distribution. Since we use a fixed number of cells per bucket, to avoid overflow requires a small average load as compared to the maximum load, leading to a lot of wasted spaced. Using d-left hashing dramatically reduces this waste.

We now explain the subtle problem with using d-left hashing directly. Let us suppose that our hash table is split into d subtables, each with B buckets. To use d-left hashing, we would naturally use a hash function $H : U \to [B]^d \times [R]$, giving d choices for each element, and store the remainder in the least loaded of the d choices (breaking ties to the left). The problem arises when it comes time to delete an element from the set. The corresponding remainder might be found in more than one of the d choices, as the same remainder might have been placed by another element in another of these d buckets at some later point in time. When this happens, we do not know which copy to delete.

It is worth making this clear by framing a specific example. Suppose that when an element x is inserted into the table, its d choices correspond to the first bucket in each subarray, and its remainder is a. Suppose further that the loads are such that the remainder is stored in the last subarray. Now suppose later than an element y is inserted into the table, its d choices correspond to the ith bucket in the ith subarray for each i, and its remainder is also a. Notice that, because the remainder a was placed in the first bucket of the *last* subarray for x, when y is placed, this remainder a will not be seen in any of y's buckets. Now suppose that, due to y, the remainder a is placed in the first bucket of the first subarray. Finally, consider what happens when we now try to delete x. The

appropriate remainder a now appears in two of x's buckets, the first and the last, and there is no way to tell which to delete. Deleting both would lead to false negatives for queries on the element y; such occurrences happen too frequently to allow this approach. Failing to delete would leave garbage in the table, causing it to fill and leading to increased false positives.

We solve this problem by breaking the hashing operations into two phases. For the first phase, we start with a hash function $H : U \to [B] \times [R]$; this gives us the true fingerprint $f_x = H(x)$ for an element. For the second phase, to obtain the d locations, we make use of additional (pseudo)-random permutations $P_1, \ldots, P_d$. Specifically, let $H(x) = f_x = (b, r)$. Then let

$$P_1(f_x) = (b_1, r_1), P_2(f_x) = (b_2, r_2), \ldots, P_d(f_x) = (b_d, r_d).$$

The values $P_i(f_x)$ correspond to the bucket and remainder corresponding to f_x for the ith subarray. Notice that for a given element, the remainder that can be stored in each subarray can be different; although this is not strictly necessary, it proves convenient for implementation. When storing an element, we first see whether in any bucket b_i the remainder r_i is already being stored. If so, we simply increment the corresponding counter. We point out that these counters can be much smaller than counters used in the standard CBF construction, as here collisions are much rarer; they occur only when H gives the same result for multiple elements. Also, as we show in Claim 3.2, only one remainder associated with f_x is stored in the table at any time, avoiding any problem with deletions. If r_i is not already stored, we store the remainder in the least loaded bucket according to the d-left scheme.

The following simple claims demonstrate the functionality of this dlCBF construction. When considering false positives below, we ignore the negligible probabilities of counter or bucket overflow, which must be considered separately.

Claim. When deleting an element in the set, only one remainder corresponding to the element will exist in the table.

Proof. Suppose not. Then there is some element $x \in S$ whose remainder is stored in subtable j to be deleted and at the same time another element $y \in S$ such that $P_i(f_x) = P_i(f_y)$ for $i \neq j$. Since the P_i are permutations, we must have that $f_x = f_y$, so x and y share the same true fingerprint. Now suppose without loss of generality that x was inserted before y; in this case, when y is inserted, the counter in subtable j associated with the remainder of x would be incremented, contradicting our assumption.

Claim. A false positive for a query z occurs if and only if $H(z) = H(x)$ for some $x \in S$.

Proof. If z gives a false positive, we have $P_i(f_x) = P_i(f_z)$ for some $x \in S$. But then $H(x) = H(z)$.

Claim. The false positive probability is $1 - (1 - 1/BR)^{|S|} \approx m/BR$.

Proof. The probability that there is no false positive for z is the probability that no $x \in S$ has $H(x) = H(z)$, and this expression corresponds to that probability.

We have thereby avoided the problem of finding two possible fingerprints to delete when handling deletions. In return, however, we have introduced another issue. Our process is no longer exactly equivalent to the d-left hashing process we have analyzed, since our d choices are no longer independent and uniform, but instead determined by the choice of the permutations. In any instantiation, there are really only BR collections of d choices available, not a full B^d. Fortunately this problem is much less significant, at least in practice. Intuitively, this is because the dependence is small enough that the behavior is essentially the same. We verify this with simulations below. A formal argument seems possible, for limited numbers of deletions, but is beyond the scope of this paper. More discussion of this point is given in the full version.

3.3 Additional Practical Issues

In practice we recommend using simple linear functions for the permutations; for example, when $H(x)$ can has range $[2^q]$, we suggest using

$$P_i(H(x)) = aH(x) \bmod 2^q$$

for a chosen uniformly at random from the odd numbers in $[2^q]$. The high order bits of $P_i(H(x))$ can be used for the bucket, and the low order bits for the fingerprint. (Because of the dependence of these hash functions, using the low order bits for the buckets is less effective; the same groups of buckets will frequently be chosen. Also, although even $H(x)$ values are then placed only in even buckets, and similarly for odd $H(x)$ values, since $H(x)$ values are (pseudo)-random the effect is negligible.) In this case, it is harder to see why the system behavior should necessarily follow the fluid limit, since the dependence among the bucket choices is quite strong with such limited hash functions. However, our simulations, discussed below, suggest the fluid limit is still remarkably accurate. (Some theoretical backing for this comes from the the recent results of [11]; again, further discussion is in the full version.)

Using simple invertible permutations P_i may allow further advantages. For example, when inserting an element, it may be possible to move other elements in the hash table, as long as each element is properly placed according to one of its d choices. (Allowing such movement of elements was the insight behind cuckoo hashing [15], and subsequent work based on cuckoo hashing, including [16].) Intuitively, such moves allow one to rectify previous placement decisions that may have subsequently turned out poorly. Recent work has shown that even limited ability to move existing elements in the hash table can yield better balance and further reduce loads [15, 16]. In particular, such moves may be useful as an emergency measure for coping with situations where the table becomes overloaded, using the moves to prevent overflow. By using simple invertible permutations P_i, one can compute f_x from a value $P_i(f_x)$, and move the fingerprint to another location given by $P_j(f_x)$. We have not studied this approach extensively, as we believe the costs outweigh the benefits for our target applications, but it may be useful in future work.

4 A Comparison with Standard Counting Bloom Filters

Roughly speaking, our experience has been that for natural parameters, our dlCBF uses half the space or less than standard CBF with the same false positive probability, and it appears as simple or even simpler to put into practice. We now formalize this comparison. Suppose, for convenience, that we are dynamically tracking a set of m elements that changes over time.

For m elements, a standard CBF using cm counters, each with four bits, as well as the theoretically optimal $k = c \ln 2$ hash functions, gives a false positive probability of approximately $(2^{-\ln 2})^c$ using $4cm$ bits. (This is slightly optimistic, because of the rounding for k.) The probability of counter overflow is negligible.

A comparable system using our dlCBF would use four subarrays, each with $m/24$ buckets, giving an average load of six elements per bucket. The method of Section 2.3 shows that providing room for eight elements per bucket should suffice to prevent bucket overflow with very high probability. Each cell counter should allow for up to four elements with the same hash value from the hash function H in the first step to prevent counter overflow with high probability. This can be done effectively with 2 bit counters, as long as one has a sentinel cell value, say 0, that cannot be a fingerprint but represents an empty cell. (We ignore the minimal effect of the sentinel value henceforth.) With an r bit remainder, the false positive probability is upper bounded by $24 \cdot 2^{-r}$, and the total number of bits used is $4m(r + 2)/3$. (For convenience, we think of $r \geq 5$ henceforth, so that our upper bound on the probability is less than 1.) Alternatively, one can think in the following terms: to obtain a false positive rate of $f = 24 \cdot 2^{-r}$, one needs to use $(4 \log_2(1/f) + 20 + 4 \log_2 3)/3$ bits per element. We note that the constant factor of $4/3$ in the leading term $4 \log_2(1/f)/3$ could be reduced arbitrarily close to 1 by using buckets with more items and filling them more tightly; the corresponding disadvantages would be a larger constant term, and more cells would need to be examined on each lookup when trying to find a matching remainder.

Equating $c = (r + 2)/3$, the two approaches use the same amount of space. But comparing the resulting false positive probabilities, we find

$$(2^{-\ln 2})^{(r+2)/3} > 24 \cdot 2^{-r}$$

for all integers $r \geq 7$. Indeed, the more space used, the larger the ratio between the false positive probability of the standard CBF and the dlCBF. For $r = 14$ and $c = 16/3$, for example, the two structures are the same size, but the false positive probability is over a factor of 100 smaller for the dlCBF. Moreover, the dlCBF actually uses less hashing than a standard CBF once the false positive probability is sufficiently small.

Alternatively, we might equalize the false positive probabilities. For example, using 9 4-bit counters (or 36 bits) per element with six hash functions in a standard CBF gives a false positive probability of about 0.01327. Using 11-bit remainders (or 52/3 bits per element) with the dlCBF gives a smaller false positive probability of approximately 0.01172. The dlCBF provides better performance with less than 1/2 the space of the standard CBF for this very natural parameter setting.

5 Simulation Results

5.1 A Full Example and Comparison

We have implemented a simulation of the dlCBF in order to test its performance and compare to a standard CBF. We focus here on a specific example, and extrapolate from it. We chose a table with 4 subarrays, each with 2048 buckets, and each bucket with 8 cells, for a total capacity of 2^{16} elements. Our target load is $3 \cdot 2^{14} = 49152$ elements, corresponding to an average load of six items per bucket. The approach of Section 2.3 suggests that bucket overload is sufficiently rare (on the order of 10^{-27} per set of elements) that it can be ignored.

We must also choose the size of the remainder and the number of counter bits per cell. In our example we have chosen 14 bit fingerprints, which as per our analysis of Section 4 should give us a false positive rate of slightly less than $24 \cdot 2^{-14} \approx 0.001465$. We use 2 bit counters per cell to handle cases where a hash value is repeated. The total size of our structure is therefore exactly 2^{20} bits.

In our construction, we use a "strong" hash function for the first phase (based on drand48), and random linear permutations exactly as described in Section 3.3 for the second phase.

For every experiment we perform, we do 10000 *trials*. In each trial, we initially begin with a set of 49152 elements, which we represent with the dlCBF; this corresponds to an average of six elements per bucket. We then simulate 2^{20} time steps, where in each time step we first delete an element from the current set uniformly at random, and then we insert a new element chosen uniformly at random from the much larger universe. This test is meant to exemplify the case where the dlCBF always remains near capacity, although the underlying set is constantly changing; it also matches the setting of our fluid limit equations. We test to make sure counter and bucket overload do not occur. After the 2^{20} time steps, we consider 10000 elements not in the final set, and see how many give false positives, in order to approximate the false positive rate that would be observed in practice.

We first consider the issue of overflow in the hash table. Over the 10000 trials, overflow never occurred. In fact, the fourth subarray *never*, over all insertions and deletions, had any buckets with eight elements, so overflow was never even an immediate danger. More concretely, we note that the fluid limit provides a very accurate representation of what we see in the simulation (even though we are using simple random linear permutations). Specifically, after all of the random insertions and deletions, we examine the bucket loads, and consider their distribution. As we can see in Table 1, the fluid limit matches the simulations extremely well, providing a very accurate picture of performance.

We now turn consider the cell counter. Recall that this counter is necessary to track when multiple elements have the same first round hash value. Over all 10000 trials, the largest this counter needed to be was 4. That is, on six of the trials, there were at some time four extant elements that shared the same hash value. This requires only two bits per cell counter (again assuming a sentinel value). While more precise calculations can be made, it is easy to

Table 1. Simulation results with $6n$ elements being placed into n buckets using four choices, compared to the differential equations. The simulation results give the fraction of buckets with load at least k for each k up to 9; the results are based on the final distribution of elements after 2^{20} deletions and insertions, averaged over 10000 trials. No bucket obtained a load of 9 at any time over all 10000 trials.

	Simulation Results	Steady State (Fluid limit)
Load ≥ 1	1.0000	1.0000
Load ≥ 2	0.9999	0.9999
Load ≥ 3	0.9990	0.9990
Load ≥ 4	0.9920	0.9920
Load ≥ 5	0.9502	0.9505
Load ≥ 6	0.7655	0.7669
Load ≥ 7	0.2868	0.2894
Load ≥ 8	0.0022	0.0023
Load ≥ 9	0.0000	1.681e-27

bound the probability of counter flow: for any set of m elements, the probability that any five will share the same hash value is at most $\binom{m}{5}\left(\frac{1}{|B||R|}\right)^5$, which for our parameters is approximately $5.62\text{e}-17$. Again, for most practical settings, counter overflow can be ignored, or if necessary a failsafe could be implemented.

Finally, we consider false positives. Over the 10000 trials, the fraction of false positives ranged from 0.00106 to 0.00195, with an average of slightly less than 0.001463. This matches our predicted performance almost exactly.

We emphasize again that this is a specific example, and the various performance metrics could be improved in various ways. False positives could be reduced by simply increasing the fingerprint size; space utilization could be improved by using fewer, larger buckets.

We now compare with a simulation of a standard CBF. We choose to compare by trying to achieve nearly the same false positive rate We use 13.5 counters per element (with 9 hash functions), or 663552 counters for 49152 elements. At four bits per counter, this works out to 2654208 bits, over 2.5 times the size of our dlCBF. Again, we performed 10000 trials, each having 2^{20} deletions and insertions after the initial insertion of the base set. The largest counter value obtained was 13, which appears in just one of the 10000 trials; counter values of 12 were obtained in 16 trials. These results match what one would obtain using a back-of-the-envelope calculation based on the Poisson approximation of the underlying balls and bins problem, as in e.g. Chapter 5 of [13] (or see also [10]). The approximate probability that a counter is hashed to by 16 elements (giving an overflow) in an optimal CBF configuration, where the expected number of hashed elements per counter is $\ln 2$, is approximately $\mathrm{e}^{-\ln 2}(\ln 2)^{16}/(16!) \approx 6.79 \cdot 10^{-17}$, roughly the same as the counter overflow probability for the dlCBF.

Over the 10000 trials, the fraction of false positives ranged from 0.00108 to 0.00205, with an average of slightly less than 0.001529. Again, this matches our

predicted performance almost exactly, and it is just slightly higher then the dlCBF. As in Section 4, our conclusion is that the dlCBF can provide the same functionality as the standard CBF, using much less space without a significant difference in complexity.

5.2 Additional Simulation Results

Suppose that we add more elements to the initial set, so that the average load per bucket is 6.5 instead of 6. (Specifically, in our simulations, we had 8192 buckets and 53428 elements, with each bucket having a capacity of 8 elements.) Using the fluid limit differential equations, we find that (in the steady state limit) the fraction of buckets with load at least 9 is approximately $2.205e-08$. We would therefore expect in our experiments, with just over 1 million deletions and insertions, that some bucket would reach a load of 9 (causing an overflow if buckets have capacity 8) slightly over 2 percent of the time. This indeed matches our simulations; over 10000 trials, we had an overflow condition in 254 trials. This example demonstrates the fact that in general the equations are quite useful for determining the threshold number of elements over which overflow is likely to occur.

These overflows could be mitigated by allowing elements to be moved, as discussed in section 3.3. We have implemented and tested this functionality as well. Specifically, we have focused on a simple improvement: if all the buckets associated with an inserted element are at capacity, we see if any of the items in just the *first* subarray can possibly be moved to another of its choices to resolve the overflow. This additional failsafe allowed us to handle an average load per bucket of 6.75 (or 55296 elements), without overflow in 10000 trials. A potential bucket overflow occured between 40 to 100 times in each trial, but even this limited allowance of movement allowed the potential overflows to be resolved. Greater loads could be handled by allowing more movement, and this is certainly worthy of further experimentation. For many applications, however, including the router-based applications we are considering, we believe that movement should remain a failsafe or a very rare special case. The small loss in space utilization is compensated for by simplicity of design.

6 Conclusion

We have demonstrated via both analysis and simulation that the dlCBF provides the same performance as a standard CBF using much less space. We believe the dlCBF will become a valuable tool in routing hardware and other products where the functionality of the counting Bloom filter is required.

One interesting area we hope to examine in future work is how to make the dlCBF responsive to large variations in load. The ability to move fingerprints offers one approach. Another interesting possibility is to dynamically changing the size of the fingerprint stored according to space needs.

A more general question relates to the use of d-left hashing. Since d-left hashing provides a natural way to obtain an "almost perfect" hash function, where else could it be used effectively to improve on a scheme that calls for perfect hashing?

References

1. Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing* 29(1):180-200, 1999.
2. P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: the heavily loaded case. In *Proc. of the 32nd Annual ACM STOC*, pp. 745–754, 2000.
3. B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.
4. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond Bloom filters: From approximate membership checks to approximate state machines. To appear in Proc. of *SIGCOMM*, 2006.
5. A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP Lookups. In *Proceedings of IEEE INFOCOM*, pp. 1454-1463, 2001.
6. A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.
7. S. Cohen and Y. Matias. Spectral Bloom Filters. *Proceedings of the 2003 ACM SIGMOD Conference*, pp. 241-252.
8. S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, 2003.
9. S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using Bloom filters. *Proceedings of the ACM SIGCOMM 2003*, pp. 201-212.
10. L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Trans. on Networking*, 8(3):281-293, 2000.
11. K. Kenthapadi and R. Panigrahy. Balanced allocation on graphs. In *Proc. of the Seventeenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 434-443, 2006.
12. Y. Lu, B. Prabhakar, and F. Bonomi. Perfect Hashing for Network Applications. To appear in *Proc. of ISIT 2006*.
13. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
14. M. Mitzenmacher and B. Vöcking. The asymptotics of selecting the shortest of two, improved. In *Analytic Methods in Applied Probability: In Memory of Fridrikh Karpelevich*, edited by Y. Suhov, American Mathematical Society, 2003.
15. R. Pagh and F. Rodler. Cuckoo Hashing. In *Proc. of the 9th Annual European Symposium on Algorithms*, pp. 121-133, 2001.
16. R. Panigrahy. Efficient hashing with lookups in two memory accesses. In *Proc. of the Sixteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 830-839, 2005.
17. A. Pagh, R. Pagh, and S. Rao. An Optimal Bloom Filter Replacement. In *Proc. of the Sixteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 823-829, 2005.
18. R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, pp. 494–505, 2005.
19. M. Sharma and J. Byers. Scalable Coordination Techniques for Distributed Network Monitoring. *6th International Workshop on Passive and Active Network Measurement (PAM)*, pp. 349-352, 2005.
20. B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the* 40^{th} *IEEE-FOCS*, pp. 131-140, 1999.