Bloom Filters via d-Left Hashing and Dynamic Bit Reassignment Extended Abstract

Flavio Bonomi[1] Michael Mitzenmacher[2] Rina Panigrahy[3] Sushil Singh[4] George Varghese[5]

***Abstract*— In recent work, the authors introduced a data structure with the same functionality as a counting Bloom filter (CBF) based on fingerprints and the d-left hashing technique. This paper describes dynamic bit reassignment, an approach that allows the size of the fingerprint to flexibly change with the load in each hash bucket, thereby reducing the probability of a false positive. This technique allows us to not only improve our d-left counting Bloom filter, but also to construct a data structure with the same functionality as a Bloom filter, including the ability to handle insertions online, that yields fewer false positives for sufficiently large filters. Our results show that our d-left Bloom filter data structure begins achieving smaller false positive rates than the standard construction at 16 bits per element. We explain the technique, describe why it is amenable to hardware implementation, and provide experimental results.**

I. Introduction

A Bloom filter is a data structure for approximating a set; it answers queries of the form is $x \in S$, with a constant probability of a false positive for elements not in the set, using a constant number of bits per element in storage and a constant number of hashes per lookup [1]. Bloom filters also naturally handle dynamic insertions of elements into the set, or equivalently they can handle the set being given as a stream of data, although generally there is a fixed maximum target size for the set. Counting Bloom filters [7], [10] allow for the deletion of elements from the set as well as insertions.

If (minimal) perfect hash functions were easy to construct, and one was given an entire set offline, one could easily obtain a data structure with the same functionality as a Bloom filter for the set [5]: use the perfect hash function to map the set of n elements into an array of size n, and store a fingerprint for each element at the corresponding location. A b-bit fingerprint would give a false positive probability of 2^{-b} with a total space of nb bits (not including the space to store the hash function). Unfortunately, such hash functions are generally difficult to construct, and are not amenable to situations where we might want to dynamically insert or delete elements into the set.

In [3], the authors use almost-perfect hash functions based on d-left hashing [11], [15] to design a data structure with the functionality of a counting Bloom filter based on this approach. (We describe the construction briefly in Section III, but we recommend [3] for more background.) For false positive rates of 1% or less, we obtained a factor of 2 or more improvement in space. A key point in the approach is that a fixed number of bits are assigned to each fingerprint, and a fixed number of fingerprint slots are assigned to each bucket in a hash table. This clearly wastes space; some slots meant to hold a fingerprint are left empty. Here we explore the idea of dynamically assigning bits in a bucket to fingerprints depending on the number of elements currently hashed to that bucket; a bucket with fewer elements would thus have longer fingerprints, reducing the overall false positive probability. We call this general approach *dynamic bit reassignment*. Dynamic bit reassignment naturally yields better performance, and it particularly improves performance when the filter is underloaded, as wasted space is the avoided. In fact, the resulting space usage is sufficiently efficient that we can provide an alternative data structure competitive with Bloom filters at a surprisingly small ratio of bits to elements. We focus on this new d-left Bloom filter structure, since an alternative Bloom filter construction may prove significant for many applications. We also consider the implications for the d-left CBF as well.

While other approaches to designing alternatives to Bloom filters have been considered (for example in [14]), we emphasize that with our approach we are sharply focused on schemes that can naturally be implemented in hardware. (Of course, they could be implemented in software as well!) That is, we are aiming for a scheme that has roughly the same implementation complexity as the very simple standard Bloom filter scheme, so that the design can be used in practice.

Finally, while we focus in this paper on variations corresponding to d-left hashing, we also believe that the idea of dynamic bit reassignment may be useful for other similar hashing techniques. Specifically, cuckoo hashing [12] and its variants [8], [13] also make use of multiple possible hash locations for an item. The main difference is that such schemes can allow much more movement of items within a hash table on an insertion, which may not be suitable for applications where sequential hash table accesses are expensive. This remains a point for future work.

II. A d-left Bloom filter construction

A. Why dynamic bit reassignment is needed

We suggest an alternative construction for a Bloom filter based on d-left hashing, which we generally refer to as a d-left Bloom filter. We use the following facts about d-left hashing, following from [4], [11]. If we use a hash

[1]Cisco Systems, Inc. bonomi@cisco.com.
[2]Division of Engineering and Applied Sciences, Harvard University. michaelm@eecs.harvard.edu. Supported in part by NSF grant CCR-0121154 and a research grant from Cisco.
[3]Stanford University. rinap@cs.stanford.edu.
[4]Cisco Systems, Inc. sushilks@cisco.com.
[5]U.C. San Diego and Cisco Systems, Inc. varghese@cs.ucsd.edu.

TABLE I

ASYMPTOTIC TAILS OF THE LOAD DISTRIBUTION FOR 3-LEFT HASH TABLES: 3 SUBTABLES, n ELEMENTS, $n/4$ TOTAL BUCKETS FOR AN AVERAGE LOAD OF 4. THE TABLE ENTRIES GIVE THE ASYMPTOTIC FRACTION OF BUCKETS WITH THE GIVEN LOAD TO TWO SIGNIFICANT DIGITS FOR LOADS UP TO 7.

Load	Fraction
0	2.3e-05
1	6.0e-04
2	1.1e-02
3	1.5e-01
4	6.6e-01
5	1.8e-01
6	2.3e-05
7	5.6e-31

table partitioned into three equally sized subtables, so that on insertion each element chooses one bucket from each subtable uniformly at random and is placed in the least loaded, then if the average load per bucket is four, the maximum load is six in practice with high probability. (The probability of overflow per bucket is, asympotically, less than 10^{-30}. See Table I.) That is, we have a set of n elements, a hash table with $n/4$ buckets divided into three subtables each with $n/12$ buckets, and each bucket need only have capacity to hold fingerprints for six elements. For n elements, we use $3n/2$ fingerprint slots. These constants are not specifically important for our scheme, but it is useful to have a specific starting point when describing the construction.

On insertion, an element is given a fingerprint, and this fingerprint is stored in the least loaded of the three associated buckets. (One can check if the fingerprint is already in one of the four buckets, in which case it need not be added.) On a lookup for an element, three buckets are searched in parallel for the appropriate fingerprint. If we use a fixed number of bits x for each bucket, divided into six slots of $f = x/6$ bits for each fingerprint, the probability of a false positive is bounded above by $12 \cdot 2^{-f}$. This follows from the fact that a false positive from any specific fingerprint is 2^{-f}, and on average an element not in the set will have 12 other elements in its buckets; a union bound now suffices. The space used for such a scheme now $3nf/2$. A corresponding Bloom filter with the same space using the optimal number of hash functions would have a false positive probability of $2^{-3f \ln 2/2} \approx 2^{-1.04f}$, better than the d-left hash table. (This follows from the standard analysis, as in [5].) While one could change the parameters used for the d-left hash table (modifying the number of subtables, the load per bucket, etc.), so that eventually it would prove superior, for natural parameter setting f must be unreasonably large in practice for the d-left hashing scheme to dominate. (With large fingerprints, say whenever $f \geq 50$, standard hashing schemes appear much more appropriate.)

Dynamic bit reassignment dramatically changes the equation, avoiding most of the wasted space from unused fingerprints, and calls for a new analysis. To see this, suppose that each bucket had 60 bits set aside for fingerprints. If fingerprint sizes are fixed, then a maximum of six fingerprints per bucket would yield 10-bit fingerprints. But suppose instead the 60 bits could be assigned dynamically, according to the number of elements present in the bucket. Buckets with four elements could then have 15-bit fingerprints, substantially improving performance. On insertion of an element in a bucket, all fingerprints in the bucket would shrink; in a bucket holding a fingerprints, each has $\lfloor 60/a \rfloor$ bits.

Our choice of 60 bits above is motivated somewhat by the fact that it is evenly divided by numbers up to six. Moreover, for each bucket we need to keep a count of how many elements are in the bucket to properly determine how bits are partitioned into fingerprints. Notice that, given this counter, partitioning bits to fingerprints is straightforward. Using four bits for a counter would give buckets a natural size of 64 bits, which might be useful in many hardware or software contexts. (While four bits for a counter is overkill, we will see how to make better use of the four bits subsequently.)

Using the rough approximation that no two fingerprints are the same (which in fact gives an upper bound on the false positive probability), we can compute the asymptotic false positive probability as n grows large by calculating the asymptotic distribution of the number of elements in a bucket for each subtable, and combining appropriately. Calculations of load distributions are described for example in [4], [11]; this is how we obtained Table I. The asymptotics are quite accurate for practical values of n. Letting Q_{ij} be the fraction of buckets in the ith table with load j, and letting $f(x)$ be the size of the fingerprint when the load is x, we find the false positive probability F is given by the formula:

$$F = \sum_a Q_{1a} a \cdot 2^{-f(a)} + \sum_b Q_{2b} b \cdot 2^{-f(b)} + \sum_c Q_{3c} c \cdot 2^{-f(c)} \quad (1)$$

That is, we consider the probability of a false positive from each table in turn, with a bucket of load a contributing $a \cdot 2^{-f(a)}$. (Again, this is a slight overestimate, as we are ignoring the possibility of duplicate fingerprints for simplicity.) Notice that, because of the tie-breaking mechanism, the Q_{ij} indeed differ among the subtables.

Performing this calculation, using 64 bits per bucket with 60 for the fingerprints and 4 bits for the counter so that $f(a) = \lfloor 60/a \rfloor$, gives a false positive rate of $0.0008937\ldots$. In comparison, a corresponding Bloom filter with 16 bits per item using the optimal 11 hash functions has a false positive rate of $0.0004587\ldots$. Our initial d-left construction is about a factor of two worse.

B. A simple improvement: semi-sorting buckets

There are many natural ways to improve our scheme, however. One particularly useful trick is to take advantage of the fact that we can re-order fingerprints in the bucket. Suppose that besides keeping a counter for each bucket, we also keep track of the number of fingerprints that begin with 0, which we call the 0-count. Then by keeping the the fingerprints in a bucket in semi-sorted order, so that all fingerprints that begin with 0 come first, we can avoid storing the first bit of each fingerprint, since this bit is given implicitly by the counters. The effect of this is to allow us to add a bit to the fingerprint without using further additional space; that is, since we have a fixed number of bits for the bucket, this space-saving improvement is used to give a longer effective fingerprint. When the count for the bucket is x, there are $x+1$ possible values for the 0-count. Hence, to track these values up to load k requires $\lceil \log_2(k(k+1)/2) \rceil$ bits.

In our setting, with 4 bits for the counter, we do not have enough bits to track the 0-count up to load 6. We can, however, use the 16 values of the counter to track the load (up to the value of 6), and track the 0-count for buckets with 4 or 5 fingerprints, which are overwhelmingly the most common cases. This gives $f(a) = 60/a$ for $a = 1, 2, 3, 6$ and $f(a) = (60/a) + 1$ for $a = 4, 5$, improving the false positive probability of our scheme to $0.0004477\ldots$, slightly better than a standard Bloom filter. (Of course there remains the possible downside of bucket overflow with our scheme, which must be handled separately or ignored.)

We note that all of the asymptotic numbers we present have also been compared with simulation numbers to verify that the analysis is correct. In all cases, the deviations from the asymptotic results are small, and are much less significant than the variance in observed false positive probabilities across individual trials. For example, we performed 1000 trials, where in each trial, we ran the corresponding balls and bins experiment with 49152 balls and 8192 bins. Recall that this slightly overestimates the false positive rate in that we ignore the possibility that two fingerprints are the same using the balls and bins analysis (but it matches the approach used for the asymptotic analysis above). After each trial, we tested for false positives on 100000 additional elements. The overall false positive rate was 0.00044988, closely matching the theoretical analysis. Over the 1000 trials, the false positives ranged from 23/100000 to 75/100000.

As a second comparison point, we consider buckets that can hold 128 bits. In this case, we use 120 bits for fingerprints, and use eight counter bits for additional purposes. An average load of 6.4 items per bucket corresponds to an average of 20 bits per item. The corresponding load distribution in the hash table is given in Table II. An optimal Bloom filter with 20 bits per item using the optimal 14 hash functions has a false positive rate of $0.00006713\ldots$. Using the same configuration for our d-left hash table, and using the counter to save an additional bit via semi-sorting for every load we obtain a false positive rate of $0.00004259\ldots$.

It turns out that we can do even better still. If we semi-sort on the first two bits, instead of just the first bit, the eight bits for the counter is sufficient to not only track the 0-count for all loads, but to also track the number of fingerprints that begin with 00, 01, 10, and 11 when the load is 6 or 7. (Indeed, with some care, we could track the number of fingerprints that begin with 00, 01, 10, and 11 when the load is 7 or 8 as well, but it turns out better to track for loads 6 and 7.) This gives $f(a) = \lfloor 120/a \rfloor + 1$ for $a \in [1, 11], a \neq 6, 7$; $f(a) = 22$ for $a = 6$, and $f(a) = 19$ for $a = 7$. These settings reduce reduces the false positive rate to $0.00002245\ldots$, almost a factor of three better than the standard Bloom filter (corresponding to roughly a 10% saving in space).

TABLE II

ASYMPTOTIC TAILS OF THE LOAD DISTRIBUTION FOR 3-LEFT HASH TABLES: 3 SUBTABLES, n ELEMENTS, $n/4$ TOTAL BUCKETS FOR AN AVERAGE LOAD OF 6.4. THE TABLE ENTRIES GIVE THE ASYMPTOTIC FRACTION OF BUCKETS WITH THE GIVEN LOAD TO TWO SIGNIFICANT DIGITS FOR LOADS UP TO 9.

Load	Fraction
0	1.7e-08
1	5.6e-07
2	1.2e-05
3	2.1e-04
4	3.5e-03
5	5.6e-02
6	4.8e-01
7	4.5e-01
8	6.2e-03
9	4.8e-15

It is worth stopping at this point to consider the implementability of the d-left scheme as we have described. We note that the hashing operations are simple; indeed, unless a method to reduce hashing (such as the one given in [9]) is used with the standard Bloom filter, the d-left scheme requires significantly less hashing than the standard Bloom filter. The d-left scheme is, by design, trivially parellizable. There is substantially more bit manipulation required using the d-left scheme, particularly in keeping the bucket semi-sorted on an insert, and when deciphering the counter bits when using semi-sorting. However, these bit operations are small in number and easily performed in hardware or software when dealing with 64 or 128 bit blocks. We therefore argue that in both hardware and software, the d-left scheme should be nearly as efficient and possibly even more efficient than a standard Bloom filter scheme.

We also note here some further points. First, our current analysis shows d-left scheme only improves on Bloom filters for fairly large, although still quite practical, numbers of bits per element. Our tradeoff point is roughly 16 bits per item, although we suspect further bit-saving tricks could drive this number down further. We suspect it will be difficult to improve on Bloom filters at false positive rates of 1% or higher, however; at such levels, Bloom filters are very close to optimal, and of course are remarkably simple. Second, while other configurations (e.g., two or four subtables) are possible, thus far our experience suggests that three subtables is best (although four is not too much worse). Third, our schemes are arguably not quite as straightforward and natural as Bloom filters. As can be when considering semi-sorting, some thinking about how to best make use of any spare bits can make a significant performance difference, while Bloom filters require much less consideration. Despite this, we believe our approach provides a quite promising alternative to Bloom filters with large numbers of bits per item.

C. A small improvement: grouping buckets

Considering equation (1), we can see that the buckets that are more full from the average contribute significantly to the false positive probability. We would like the distribution of elements to bins to be as smooth as possible, so that all elements have roughly the same long fingerprint.

One alternative way to achieve this is to group buckets together, essentially allowing more full buckets to swipe bits from less full buckets. We emphasize that this does not mean that we are creating larger buckets with more elements per bucket; we are merely organizing buckets in such a way that bits are divided among fingerprints in a group of buckets, instead of among fingerprints in a single bucket. An advantage of this approach is that is also can be quite sensible in hardware: for example, if a memory read obtains 256 bits, it would be natural to group four 64-bit buckets together into a single memory block. Again, this somewhat complicates the bit-level operations that must be taken on a lookup and an insert, but not dramatically.

As an example, using our example of 64-bit buckets with 60 bits set aside for fingerprints, if four collected buckets had loads 5, 4, 4, and 3, all elements could be given 15 bit fingerprints. If the four buckets had load 5, 5, 4, and 4, then thirteen elements (three buckets) could be given thirteen bit fingerprints, and five elements (one bucket) could be given fourteen bit fingerprints. Such smoothing offers small but non-trivial gains.

Again, the asymptotics can be calculated exactly via differential equations. To simplify our calculations, we restrict each bucket within a group to have the same number of bits, even though the example of loads 5-5-4-4 given above shows that this can be wasteful if after such a division there are enough bits left over to give one extra bit to every fingerprint in just some of the buckets in the group. As an example, with our 64-bit buckets, grouping together blocks of four buckets in the same subtable (and using semi-sorting as above) drops the false positive rate from $0.0004477\ldots$ to $0.0003520\ldots$. With our 128-bit buckets, such grouping reduces the false positive rate from $0.00002245\ldots$ to $0.0002042\ldots$.

D. Further possible improvements: balancing

In our work up to this point, we have implicitly assumed that elements to be inserted into the d-left Bloom filter are presented one at a time and the corresponding fingerprint must be placed immediately without being moved subsequently. This makes sense when elements are presented as a stream of data, as they are for many applications. The setting is different if all the elements are presented initially offline, and one is given time offline to build the data structure.

In the setting of d-left Bloom filters, we could determine their performance in the offline setting, where we should be able to balance the load among buckets much more effectively. This can be done by various means, including for example local search. We have done this experimentally, and not surprisingly found sizable improvements. Of course, in the offline case, we could do even better by simply looking for a minimal perfect hash function, as described in the introduction. Alternatively, and more efficiently than finding a minimal perfect hash function, we could use a variant of cuckoo hashing: using buckets of size 1, with high probability we can place n elements into less than $1.05n$ cells, with four possible locations for each element given by four distinct hash functions, in the offline setting [8]. Thus cuckoo hashing gives something very close to a minimal perfect hash function, using slightly more space and using more than one hash function. (While natural, this use of cuckoo hashing, to construct a near-optimal Bloom filter offline, does not seem to have been explicitly mentioned in previous work.)

The idea of re-balancing, however, can still be useful in some situations. First, if one is can delay insertions and handle them in large batches, one can then balance the batches in an offline fashion. Second, re-balancing operations can actually be done online or as a background task if one can use the fingerprint and bucket for an element in one subtable to determine the appropriate fingerprints and buckets for an element in another subtable. This would be possible, for example, using invertible pseudorandom permutations, or simply linear permutations, as described in [3] and below. Such invertible hashes would also potentially allow using cuckoo hashing to construct Bloom filters online. A problem with this approach is that we do not know how to analyze performance when limited to such hash functions, except by simulation. (In practice, behavior generally appears very close to the idealized analysis of fully random hash functions.)

To gauge the potential utility of these approaches, we summarize our experiments on the performance of offline assignments. The Bloom filter is set up by initiall inserting all items and then sequentially trying to replace each element in order 10 times after the initial insertion. We use semi-sorting to improve performance, but we do not group buckets. For 32768 elements, using two hash functions and subtables with 4096 buckets each (for an average load of 4), we obtain an average false positive rate of 0.00023894 when using 64-bit buckets. This is a fairly small improvement, but the initial setup was already quite well balanced. For 49152 elements, using two hash functions and subtables with 4096 buckets each (for an average load of 6), we obtain an average false positive rate of 0.000004605 when using 128-bit buckets. This demonstrates a more substantial improvement.

E. A consideration: underloaded filters

Overloading is a problem for both standard Bloom filters and our d-left variation; besides yielding increased false positive, with our d-left Bloom filter there is a danger of bucket overflow. But it is also worth considering performance when the filters are underloaded. In many settings, as one inserts items on-line, one also asks lookup queries, before the filter has reached its target full state. For example, in a model checking application described in [6], states are inserted into a Bloom filter, and as new states are encountered they are checked to see if they are already in the Bloom filter. One performance metric is the number of false positives as the

Bloom filter is filled. For most of its lifetime, the Bloom filter is underloaded.

Our d-left scheme performs quite well when underloaded. This is not surprising; again, as long as the bits in a bucket are being used to provide as long a fingerprint as possible, one would expect good performance. Details depend on the configuration, but are similar to our previous experiments. We point out one important issue. Because semi-sorting is quite useful, not having semi-sorting available for all loads can be damaging. For example, in our 64-bit bucket configuration, we describe enabling semi-sorting only when the load is four or five. When the average load is four, this is suitable; when the filter is significantly underloaded, so that the average load is much less than four, then we get no gain from semi-sorting under this configuration.

F. Result summary

We have demonstrated that the d-left hashing approach, when combined with dynamic bit reassignment within a bucket and the bit-saving trick of semi-sorting, can outperform a standard Bloom filter, starting at roughly 16 bits per element. Further gains are possible by applying additional techniques to save bits wherever possible, but thus far we have found these techniques generally give only small gains in the false positive probability for potentially non-trivial increases in computational requirements. The basic approach is comparable with the complexity of a standard Bloom filter, and certainly appears viable in practice, in both hardware and software.

III. Counting Bloom filter variant

A. Extending the Previous Analysis

Dynamic bit reassignment can also improve the performance of the proposed d-left counting Bloom filter structure [2], [3]. Again, instead of wasting space statically assigning a fixed number of bits for each fingerprint, we allow bits in a bucket to be assigned dynamically, yielding a dynamic d-left counting Bloom filter, or ddlCBF. A further advantage of dynamic bit reassignment is that is can possibly allow a more a graceful response to overload conditions; instead of facing overflow in the buckets, one merely shortens the fingerprints, increasing the false positives but avoiding overflow.

We must take some care in order to ensure that we avoid any problems with regard to deletion. We therefore explain in detail the hashing and placement scheme. Knowledge of [3] is useful for understanding these results. In the setting we now describe, we assume that the number of bits per fingerprint stored in any bucket can *only decrease over time*. We return to this point later. Following the notation of [3], we refer to fingerprints as *remainders* in this section. Each bucket must contain a counter for the *current* number of remainders stored the bucket, as well as the *maximum* number of remainders ever stored in the bucket; the latter determines the current size of the store remainder. In [3], each *cell*, or remainder location, also has a small counter associated with it (in case remainders match). We assume this is used in what follows for analysis purposes. An alternative approach would be to just include multiple copies corresponding to the same remainder; where appropriate we briefly discuss this alternative as well.

As with the original d-left CBF, we break the hashing operations down into two phases. For the first phase, we start with a hash function $H : U \rightarrow [B] \times [R] \times [Z]$. We write $H(x) = f_x = (b, r, z)$. The intuition here is that the first two fields b and r will specify the bucket and a *minimal remainder* for each cell. For the second phase, to obtain the d locations for an element, we make use of additional (pseudo)-random permutations $P_1, \ldots, P_d$, where $P_i : [B] \times [R] \rightarrow [B] \times [R]$. Specifically, let $F(H(x)) = (b, r)$. Then let

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

The values $P_i(F(H(x)))$ correspond to the bucket and minimal remainder corresponding to x for the ith subarray. Notice that for a given element, the minimal remainder that can be stored in each subarray can be different.

The minimal remainder must always be stored for each element, as it is used to avoid any issues of ambiguity when a deletion occurs, following the approach of [3]. Notice that this means that the ddlCBF still cannot handle sets of an arbitrary size (as a standard CBF potentially could). The z field simply provides extra remainder bits that can be stored as room permits. Generally Z will be such so that the maximum length remainder is bounded to some reasonable size, such as 32 or 64.

When inserting an element, we first see whether in any bucket b_i another element with the same minimal remainder r_i is already being stored. If so, we assume the element is placed in the same bucket, either by incrementing the counter (if the two elements match in all the bits stored in the bucket after placement) or by simply placing as much of the remainder bits r_i, z as possible given the load of the bucket. (Again, multiple copies could be placed instead of using counters, and copies can be placed in any bucket.) If no appropriate r_i is already stored, we store the remainder in the least loaded bucket according to the d-left scheme. On an insertion, all fingerprints are reduced in size as appropriate.

On a deletion, the remainder that matches an element is deleted or the corresponding counter decremented. If counters are not being used, the remainder that gives the longest match should be deleted; using the longest match on a deletion maintains consistency over future deletions for lookups.

With this setup, adding dynamic bit reassignment does not essentially change the behavior of the dlCBF, except that the permutations are now based on the minimal remainder. In particular, if one uses a minimal remainder size corresponding to the remainder size of a non-dynamic dlCBF, the performance can only improve (if one ignores the space cost of the counters). That is, a ddlCBF will only yield remainders of size less than the minimal remainder when the corresponding dlCBF would have overflowed. It is worth noting that assuming we always place a fingerprint in a bucket that already contains the minimal remainder, the underlying process is not exactly that of d-left hashing in general. However, since all elements with the same minimal remainder would collapse into a single cell (with a corresponding count in the counter) if a bucket becomes loaded to the point where the remainder stored was minimal, the behavior is again no worse that the standard dlCBF with remainder size equal to the corresponding minimal remainder. (If not using counters, then there is some slight dependence among bucket choices, since elements with the same initial hash will choose the same bucket. Enough slack must be left for such elements, since they are no longer handled by counters.)

B. A small improvement: multiple remainder sizes

In the basic description we have provided thus far, the remainder size in a bucket depends on the maximum number of elements that have ever been in a bucket. Obviously, eventually all buckets will obtain a larger than average number of items, and the scheme will converge in performance to that of the dlCBF without dynamic bit reassignment. With proper engineering, the advantage of dynamic bit reassignment can last for a significant time, but it is somewhat disappointing that it is not permanent. One approach to handle this would be to keep an additional (off-chip) structure that held larger versions of the remainders, which could be used to refresh short remainders periodically.

Another approach is to allow multiple remainder lengths within a bucket. After items are deleted, freeing up space in a bucket, newly entering items try to take as much space as possible. That is, remainders that are below average in size "donate" bits to the newer remainders as they are inserted, so all bits are being used even though older remainders cannot grow in length. In this way, space is reclaimed eventually from temporary overloading, although the time for older, smaller remainders to be removed can vary significantly. These shorter remainders will be the bottleneck in terms of the false positive rate.

The main downside of this approach is that handling the bit-level manipulations for a bucket becomes somewhat more complex. It is not clear that the small gains from this approach adequately cover the additional cost in terms of bit operations per table operation and complexity of implementation.

C. Experimental results

In order to gain some preliminary idea of how much improvement we could expect from dynamic bit reassignment, we recreated the experiment from the paper [3]. Recall that there it was shown that a dlCBF could perform more than a factor of two better in terms of space with the same false positive rate.

We use a similar construction as in that experiment: 4 subtables, with 2048 buckets in each subtable, and an average load of six elements per bucket. Similarly, as in that experiment, we start by inserting 49152 elements, and then we repeatedly delete a random element and insert a new random element 2^{20} times. (We used a slightly different construction in this setting, using 1 bit counters per cell instead of 2 bit counters per cell, and otherwise we allow remainders to appear multiple times. This had no substantial effect on our results.) The false positive rate was measured after all of these insertions and deletions.

There are slight variations in the final sizes of our structures, depending on the number of bits used in the bucket counters, but all structures use 120 bits for storing remainders and are about 1 Mbit in size. We averaged over fifty trials.

For the dlCBF scheme, our false positive rate was 0.0014593, matching quite closely the results reported in [3]. For a ddlCBF scheme using only one remainder length for all items in a bucket (based on the maximum load the bucket has seen) using semi-sorting, the false positive rate was 0.0004729, roughly a factor of three better. This experimental result matches calculations obtained by considering the experimentally determined distribution of the maximum load seen over the buckets. The key point is that most buckets have had a maximum load of only seven, instead of eight, giving the performance gain. We emphasize that this performance comparison would have been even better for the ddlCBF if fewer random deletion/insertion operations had been done, since the fingerprint size from the ddlCBF is determined by the maximum load a bucket has seen. Similarly, comparative performance would have been better if the system had been underloaded, since cells are fixed size in the dlCBF scheme but variable sized in the ddlCBF scheme. Finally, we also did an experiment where each bucket was allowed two fingerprint sizes. This approach did use slightly more space (semi-sorting was not used as it was not cost-effective using two fingerprint sizes), and reduced the false positive rate to 0.0003111. The improvement over the initial ddlCBF scheme was small; performance would be comparatively better if the system were overloaded, but generally with these structures we seek to avoid overloading.

IV. Conclusions

The use of dynamic bit reassignment is extremely natural for hardware implementations of Bloom filters and related data structures, as data is usually accessed in standard block sizes, and bit-level manipulations can be performed effectively. We have demonstrated the effectiveness of dynamic bit reassignment to design new data structures with the functionality of a Bloom filter, and to improve our previous dlCBF design.

There still is room to improve our scheme to get close to the theoretical lower bounds. Specifically, in examining how our approach differs from perfect hashing, we see that the overhead of having to look in more than one bucket and at multiple fingerprints per bucket translates directly into a nontrivial constant factor multiplier for the false positive rate. This factor might be removed in some settings by using cuckoo hashing techniques [12], [8], [13], although the issue of hvaing to move elements with that approach must be considered. Is there some way to remove the factor entirely with computationally simple techniques?

References

[1] B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.

[2] 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.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. To appear in *Proc. of ESA*, 2006.

[4] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP Lookups. In *Proceedings of IEEE INFOCOM*, pp. 1454-1463, 2001.

[5] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.

[6] P. Dillinger and P. Manolios. Fast *and* Accurate Bitstate Verification for SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, 2004.

[7] 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*, 8(3):281-293, 2000.

[8] D. Fotakis, R. Pagh, P. Sander, and P. Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proceedings of the Twentieth Annual Symposium on Theoretical Aspects of Computer Science*, pp. 271-282, 2003.

[9] A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Multiple Choices. In *Proceedings of the Forty-Third Annual Allerton Conference on Communication, Control, and Computing*, 2005.

[10] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002.

[11] 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.

[12] R. Pagh and F. Rodler. Cuckoo Hashing. In *Proc. of the 9th Annual European Symposium on Algorithms*, pp. 121-133, 2001.

[13] R. Panigrahy. Efficient hashing with lookups in two memory accesses. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 830-839, 2005.

[14] A. Pagh, R. Pagh, and S. Srinivas Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 823-829, 2005.

[15] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the* 40^{th} *IEEE-FOCS*, pp. 131-140, 1999.