Compressed Bloom Filters

Michael Mitzenmacher, *Member, IEEE*

***Abstract*—A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Although Bloom filters allow false positives, for many applications the space savings outweigh this drawback when the probability of an error is sufficiently low. We introduce *compressed Bloom filters*, which improve performance when the Bloom filter is passed as a message, and its transmission size is a limiting factor. For example, Bloom filters have been suggested as a means for sharing Web cache information. In this setting, proxies do not share the exact contents of their caches, but instead periodically broadcast Bloom filters representing their caches. By using compressed Bloom filters, proxies can reduce the number of bits broadcast, the false positive probability, and/or the amount of computation per lookup. The cost is the processing time for compression and decompression, which can use simple arithmetic coding, and more memory use at the proxies, which utilize the larger uncompressed form of the Bloom filter.**

***Index Terms*—Algorithms, computer networks, information theory, distributed computing, distributed information systems.**

I. Introduction

BLOOM filters are an excellent data structure for succinctly representing a set in order to support membership queries [3]. We describe them in detail in Section II-A; here, we simply note that a Bloom filter is a bit array, it is randomized (in that it uses randomly selected hash functions), and it has some probability of leading to a *false positive*, that is, it may posit that an element is in a set when it is not. For many applications, the probability of a false positive can be made sufficiently small and the space savings are significant enough that Bloom filters are useful.

In fact, Bloom filters have a great deal of potential for distributed protocols where systems need to share information about what data they have available. For example, Fan *et al.* describe how Bloom filters can be used for Web cache sharing [7], [18]. To reduce message traffic, proxies do not transfer URL lists corresponding to the exact contents of their caches, but instead periodically broadcast Bloom filters that represent the contents of their caches. If a proxy wishes to determine if another proxy has a page in its cache, it checks the appropriate Bloom filter. In the case of a false positive, a proxy may request a page from another proxy, only to find that that proxy does not actually have that page cached. In that case, some additional delay has been incurred. The small chance of a false positive introduced by using a Bloom filter is greatly outweighed by the significant reduction in network traffic achieved by using the succinct Bloom filter instead of sending the full list of cache contents. This technique is used in the open source Web proxy cache Squid, where the Bloom filters are referred to as cache digests [16], [14]. Bloom filters have also been suggested for other distributed protocols, e.g., [6], [10], [15].

Our paper is based on the following insight. In this situation, the Bloom filter plays a dual role. It is both a data structure being used at the proxies, and a message being passed between them. When we use the Bloom filter as a data structure, we may tune its parameters for optimal performance as a data structure. For example, given a memory size (or more specifically, the number of bits allowed in the bit array that represents the Bloom filter) and the number of elements in the set to be represented, we can minimize the probability of a false positive. Indeed, this is the approach taken in the analysis of [7], [18]. If this data structure is also being passed around as a message, however, then we introduce another performance measure we may wish to optimize for: transmission size. The transmission size is the size of the data being sent; if no compression is used, it is simply the size of the bit array, but it could potentially be smaller once compression is introduced. Transmission size may be of greater importance when the amount of network traffic is a concern but there is memory available at the endpoint machines. This is especially true in distributed systems where information must be transmitted repeatedly from one endpoint machine to many others. For example, in the Web cache sharing system described above, the required memory at each proxy is linear in the number of proxies, while the total message traffic rate is quadratic in the number of proxies, assuming point-to-point communication is used. Moreover, the amount of memory required at the endpoint machines is fixed for the life of the system, where the traffic is additive over the life of the system.

In this paper, we show how compressing a Bloom filter can lead to improved performance. By using compressed Bloom filters, protocols reduce the number of bits broadcast, the false positive probability, and/or the amount of computation per lookup. The tradeoff costs are the increased processing requirement for compression and decompression and larger memory requirements at the endpoint machines, which may use a larger original uncompressed form of the Bloom filter in order to achieve improved transmission size.

We start by defining the problem as an optimization problem, which we solve using some simplifying assumptions. We then consider practical issues, including effective compression schemes and actual performance. We recommend arithmetic coding [12], a simple compression scheme well suited to this situation with fast implementations. We follow by showing how to extend our work to other important cases, such as in

Manuscript received August 1, 2001; revised December 5, 2001; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Rexford. This work was supported in part by the National Science Foundation under CAREER Grant CCR-9983832, Grant CCR-0118701, Grant CCR-0121154, and an Alfred P. Sloan Research Fellowship.

The author is with Harvard University, Cambridge, MA 02138 USA (e-mail: michaelm@eecs.harvard.edu).

Digital Object Identifier 10.1109/TNET.2002.803864.

the case where it is possible to update by sending changes (or deltas) in the Bloom filter rather than new Bloom filters.

Our work underscores an important general principle for distributed algorithms: when using a data structure as a message, one should consider the parameters of the data structure with both of these roles in mind. If transmission size is important, tuning the parameters so that compression can be used effectively may yield dividends.

II. Compressed Bloom Filters: Theory

A. Bloom Filters

We begin by introducing Bloom filters, following the framework and analysis of [7], [18].

A Bloom filter is used to represent a set $S = \{s_1, s_2, \ldots, s_n\}$ of n elements from a universe U. Note that representing elements of the underlying universe uniquely with fixed length identifiers requires $\lceil \log |U| \rceil$ bits per element, so transmitting the set directly requires $n\lceil \log |U| \rceil$ bits. A Bloom filter consists of an array of m bits, initially all set to 0; generally m/n is a fixed constant determined by design for the application. A Bloom filter uses k independent random hash functions $h_1, \ldots, h_k$ with range $\{0, \ldots, m-1\}$. We make the natural assumption that these hash functions map each element in the universe to a random number uniform over the range $\{0, \ldots, m-1\}$ for mathematical convenience. For each element $s \in S$, the bits $h_i(s)$ are set to 1 for $1 \leq i \leq k$. A location can be set to 1 multiple times, but only the first change has an effect. To check if an element x is in S, we check whether all $h_i(x)$ are set to 1. If not, then clearly x is not a member of S. If all $h_i(x)$ are set to 1, we assume that x is in S, although we are wrong with some probability. Hence, a Bloom filter may yield a *false positive*, where it suggests that an element x is in S even though it is not. For many applications, this is acceptable as long as the probability of a false positive is sufficiently small.

The probability of a false positive for an element not in the set, or the *false positive probability*, can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random. After all the elements of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \approx \mathrm{e}^{-kn/m}.$$

We let $p = \mathrm{e}^{-kn/m}$. For a false positive to occur, when an element not in the set is checked, each of the k locations checked must not contain a 0. To simplify the analysis, let us assume that entries in the Bloom filter are independently set to 0 with probability p and set to 1 with probability $1 - p$. (Technically, this is not precisely true, both because the fraction of bits set to 1 is a random variable and the bits are not completely independent: the fact that one bit was set to 1 affects the probability of other bits being set to 1, since the set element and hash that set a bit to 1 cannot set any other bit to 1. An argument showing that the fraction of 0 entries is sharply concentrated around p is given in the Appendix. Also, asymptotically and in practice the dependence is negligible; see, for example, [1]. In fact, independence is not required for the argument below, but henceforth, we make the simplifying assumption of independence for ease of exposition.) The probability of a false positive is thus

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - \mathrm{e}^{-kn/m}\right)^k = (1-p)^k.$$

We let $f = (1 - \mathrm{e}^{-kn/m})^k = (1-p)^k$. From now on, for convenience, we use the asymptotic approximations p and f to represent, respectively, the probability that a bit in the Bloom filter is 0 and the probability of a false positive.

Although it is clear from the above discussion, it is worth noting that there are *three* fundamental performance metrics for Bloom filters that can be traded off: 1) computation time (corresponding to the number of hash functions k); 2) size (corresponding to the array size m); and 3) the probability of error (corresponding to the false positive probability f).

Suppose we are given m and n and we wish to optimize the number of hash functions k to minimize the false positive probability f. There are two competing forces: using more hash functions gives us more chances to find a 0 bit for an element that is not a member of S, but using fewer hash functions increases the fraction of 0 bits in the array. The optimal number of hash functions that minimizes f as a function of k is easily found taking the derivative. More conveniently, note that f equals $\exp(k \ln(1 - \mathrm{e}^{-kn/m}))$. Let $g = k \ln(1 - \mathrm{e}^{-kn/m})$. Minimizing the false positive probability f is equivalent to minimizing g with respect to k. We find

$$\frac{dg}{dk} = \ln\left(1 - \mathrm{e}^{-(kn/m)}\right) + \frac{kn}{m} \frac{\mathrm{e}^{-(kn/m)}}{1 - \mathrm{e}^{-(kn/m)}}.$$

It is easy to check that the derivative is 0 when $k = (\ln 2) \cdot (m/n)$; further efforts reveal that this is a global minimum. In this case the false positive probability f is $(1/2)^k = (0.6185)^{m/n}$. In practice, of course, k must be an integer, and smaller k might be preferred since they reduce the amount of computation necessary.

For comparison with later results, it is useful to frame the optimization another way. Letting f be a function of p, we find

$$\begin{aligned} f &= (1-p)^k \\ &= (1-p)^{(-\ln p) \cdot (m/n)} \\ &= \left(\mathrm{e}^{-\ln(p) \cdot \ln(1-p)}\right)^{m/n}. \end{aligned} \tag{1}$$

From the symmetry of this expression, it is easy to check that $p = 1/2$ minimizes the false positive probability f. Hence, the optimal results are achieved when each bit of the Bloom filter is 0 with probability (roughly) 1/2.

Note that Bloom filters are highly effective even if $m = cn$ for a small constant c, such as $c = 8$. The obvious approach when more bits are available is to simply hash each element into $\Theta(\log n)$ bits and send a list of hash values. Bloom filters can allow significantly fewer bits to be sent while still achieving a very good false positive probability.

B. Compressed Bloom Filters

Our optimization above of the number of hash functions k is based on the assumption that we wish to minimize the failure of a false positive as a function of the array size m and the number of objects n. This is the correct optimization if we consider the Bloom filter as an object residing in memory. In the Web

cache application, however, the Bloom filter is not just an object that resides in memory, but an object that must be transferred between proxies. This fact suggests that we may not want to optimize the number of hash functions for m and n, but instead optimize the number of hash functions for the size of the data that needs to be sent, or the *transmission size*. The transmission size, however, need not be m; we might be able to compress the bit array. Therefore, we choose our parameters to minimize the failure probability after using compression.

Let us consider the standard uncompressed Bloom filter, which is optimized for $k = (\ln 2) \cdot (m/n)$, or, equivalently, for $p = 1/2$. Can we gain anything by compressing the resulting bit array? Under our assumption of good random hash functions, the bit array appears to be a random string of m 0's and 1's, with each entry being 0 or 1 independently with probability 1/2.[1] Hence, compression does not gain anything for this choice of k.

Suppose, however, we instead choose k so that each of the entries in the m bit array is 1 with probability 1/3. Then we can take advantage of this fact to compress the m bit array and reduce the transmission size. After transmission, the bit array is decompressed for actual use. Note that the uncompressed Bloom filter size is still m bits. While this choice of k is not optimal for the uncompressed size m, if our goal is to optimize for the transmission size, using compression may yield a better result. The question is whether this compression gains us anything, or if we would have been better off simply using a smaller number of bits in our array and optimizing for that size.

We assume here that all lookup computation on the Bloom filter is done after decompression at the proxies. A compression scheme that also provided random access might allow us to compute on the compressed Bloom filter; however, achieving random access, efficiency, and good compression simultaneously is generally difficult. One possibility is to split the Bloom filter into several pieces, and compress each piece. To look up a bit would only require decompressing a certain piece of the filter instead of the entire filter, reducing the amount of memory required [11]. This approach will slightly reduce compression but greatly increase computation if many lookups occur between updates.

To contrast with the original Bloom filter discussion, we note that for compressed Bloom filters there are now *four* fundamental performance metrics for Bloom filters that can be traded off. Besides computation time (corresponding to the number of hash functions k) and the probability of error (corresponding to the false positive probability f), there are two other metrics: the *uncompressed filter size* that the Bloom filter has in the proxy memory, which we continue to denote by the number of array bits m; and the *transmission size* corresponding to its size after compression, which we denote by z. Our starting point is the consideration that in many situations the transmission size may be more important than the uncompressed filter size.

We may establish the problem as an optimization problem as follows. Let z be the desired compressed size. Recall that each bit in the bit array is 0 with probability p; we treat the bits as independent. Also, as a mathematically convenient approximation, we assume that we have an optimal compressor. That is, we assume that our m bit filter can be compressed down to only $mH(p)$ bits, where $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$ is the binary entropy function. Our compressor, therefore, uses the optimal $H(p)$ bits on average for each bit in the original string. We consider the practical implications more carefully subsequently. Here, we note just that near-optimal compressors exist; arithmetic coding, for example, requires on average less than $H(p) + \epsilon$ bits per character for any $\epsilon > 0$, given suitably large strings.

Our optimization problem is as follows: given n and z, choose m and k to minimize f subject to $mH(p) \leq z$. One possibility is to choose $m = z$ and $k = (\ln 2) \cdot (m/n)$ so that $p = 1/2$; this is the original optimized Bloom filter. Hence, we can guarantee that $f \leq (0.6185)^{z/n}$.

We can, however, do better. Indeed, in theory this choice of k is the *worst* choice possible once we allow compression. To see this, let us again write f as a function of p: $f = (1-p)^{(-\ln p)\cdot(m/n)}$ subject to $m = z/H(p)$ (we may without loss of generality choose m as large as possible). Equivalently, we have

$$f = (1-p)^{-(z\ln p/nH(p))}.$$

Since z and n are fixed with $z \geq n$, we may equivalently seek to minimize $\beta = f^{n/z}$. Simple calculations show

$$\begin{aligned}\beta &= (1-p)^{-(\ln p/H(p))} \\ &= \mathrm{e}^{-(\ln p\cdot\ln(1-p)/H(p))} \\ &= \exp\left(\frac{-\ln(p)\cdot\ln(1-p)}{(-\log_2 \mathrm{e})(p\ln p + (1-p)\ln(1-p))}\right). \quad (2)\end{aligned}$$

It is interesting to compare this equation with (1); the relevant expression in p shows a similar symmetry, here with additional terms due to the compression.

The value of β is maximized when the exponent is maximized, or equivalently when the term

$$\gamma = \left(\frac{p}{\ln(1-p)} + \frac{1-p}{\ln p}\right)$$

is minimized. Note that

$$\frac{d\gamma}{dp} = \frac{1}{\ln(1-p)} - \frac{1}{\ln p} + \frac{p}{(1-p)\ln^2(1-p)} - \frac{1-p}{p\ln^2(p)}.$$

The value of $d\gamma/dp$ is clearly 0 when $p = 1/2$, and using symmetry it is easy to check that $d\gamma/dp$ is negative for $p < 1/2$ and positive for $p > 1/2$. Hence, the maximum probability of a false positive using a compressed Bloom filter occurs when $p = 1/2$, or equivalently $k = (\ln 2) \cdot (m/n)$.

We emphasize the point again: the number of hash functions that minimizes the false positive probability without compression in fact maximizes the false positive probability with compression. Said in another way, in our idealized setting using compression always decreases the false positive probability.

The argument above also shows that γ is maximized and, hence, β and f are minimized in one of the limiting situations as p goes to 0 or 1. In each case, using, for example, the expansion $\ln(1-x) \approx -x - x^2/2 - x^3/3 - \ldots$, we find that γ goes to -1. Hence, β goes to 1/2 in both limiting cases, and we can in theory

[1]Again, technically the bits are not completely independent, but they are so near independent that the difference is unimportant for this argument.

achieve a false positive probability arbitrarily close to $(0.5)^{z/n}$ by letting the number of hash functions go to 0 or infinity.

It is an interesting and worthwhile exercise to try to understand intuitively how the expression $f = (0.5)^{z/n}$ for the limiting case arises. Suppose we start with a very large bit array, and use just one hash function for our Bloom filter. One way of compressing the Bloom filter would be to simply send the array indices that contain a 1. Note that this is equivalent to hashing each element into a z/n bit string; that is, for one hash function and suitably large values of z and m, a compressed Bloom filter is equivalent to the natural hashing solution. Thinking in terms of hashing, it is clear that increasing the number of bits each element hashes into by 1 drops the false positive probability by approximately 1/2, which gives some insight into the result for Bloom filters.

In practice, we are significantly more constrained than the limiting situations suggest, since, in general, letting p go to 0 or 1 corresponds, respectively, to using an infinite number of one or zero hash functions. Of course, we must use at least one hash function! Note, however, that the theory shows we may achieve improved performance by taking $k < \ln 2 \cdot (m/n)$ for the compressed Bloom filter. This has the additional benefit that a compressed Bloom filter uses fewer hash functions and, hence, requires less computation per lookup. This contrasts with the additional computation required for encoding before transmission and decoding upon receipt, which are one-time costs. Further practical considerations are discussed in Section III.

The optimization framework developed above is not the only one possible. For example, one could instead fix the desired false positive probability f and optimize for the final compressed size z. To compare in this situation, note that in the limit as the number of hash functions goes to 0, the compressed Bloom filter has a false positive probability tending to $(0.5)^{z/n}$, while the standard Bloom filter has a false positive probability tending to $(0.5)^{(m \ln 2)/n}$. Hence, the best possible compressed Bloom filter achieving the same false positive probability as the standard Bloom filter would have $z = m \ln 2$, a savings in size of roughly 30%. Again, this is significantly better than what can be realized in practice.

The primary point of this theoretical analysis is to demonstrate that compression is a viable means of improving performance, in terms of reducing the false positive probability for a desired compressed size, or for reducing the transmission size for a fixed false positive probability. Indeed, because the compressed Bloom filter allows us another performance metric, it provides more flexibility than the standard original Bloom filter. An additional benefit is that the compressed Bloom filters use a smaller number of hash functions, so that lookups are more efficient. Based on this theory, we now consider implementation details and specific examples.

III. Compressed Bloom Filters: Practice

Our theoretical analysis avoided several issues that are important for a real implementation:

- *Restrictions on m*: While the size z of the compressed Bloom filter may be of primary importance, limitations on the size m of the uncompressed Bloom filter also constrain the possibilities. For example, while theoretically we can do well using one hash function and compressing, achieving a false positive probability of ϵ with one hash function requires $m \approx n/\epsilon$, which may be too large for real applications.

Also, it may be desirable to have m be a power of two for various computations. We do not restrict ourselves to powers of two here.

- *Compression overhead*: Compression schemes do not achieve optimal performance; all compression schemes have some associated overhead so that they do not exactly match the space as given by the entropy formula. Hence, the gain from the compressed Bloom filter must overcome the associated overhead costs.
- *Compression variability*: Of perhaps greater practical importance is that if there is an absolute maximum packet size, we generally want that the compressed array does not exceed the size of some fixed number of packets. Compression performance, however, varies depending on the input. We would like to make our Bloom filter memory size m as large as possible while maintaining a high probability that the compressed size z does not exceed a given threshold, so that we do not send additional packets beyond the threshold with little information. A related problem is that if the number of elements n in the set S cannot be determined in advance, a misprediction of n could yield insufficient compression.
- *Hashing performance*: Depending on the data and the hash functions chosen, real hash functions may behave differently from the analysis above.

The issue of achieving good hashing performance on arbitrary data sets is outside the scope of this paper, and we do not consider it further except to raise the following points. First, in practice we suspect that using standard universal families of hash functions [5], [13] or MD5 (used in [7], [18]) will be suitable. Second, in situations where hashing performance is not sufficiently random, we expect that compressed Bloom filters will still generally outperform the uncompressed Bloom filter. The point is that if the false positive probability of a compressed Bloom filter is increased because of weak hash functions, we would expect the false positive probability of the uncompressed Bloom filter to increase as well; moreover, since compressed Bloom filters use fewer hash functions, we expect the effect will be worse for the uncompressed filter. For compressed Bloom filters, however, there is the additional problem that weak hash functions may yield bit arrays that do not compress as much as expected. The choice of parameters may, therefore, have to be tuned for the particular data type. Finally, for the Bloom filter to function properly, all of the senders and receivers must agree on the hash functions used. There are several ways to achieve agreement; the correct approach may depend on the application. The hash functions can be fixed once and for all. With this approach, in rare instances a specific data set may yield poor performance, in that it might not compress as well as expected or the false positive rate might be higher than expected. Alternatively, the sender can specify the hash functions from the family to be used in some explicit form with the Bloom filter. This approach incurs some overhead, but it allows the sender to avoid hash functions that perform poorly on specific data. The hash

TABLE I
AT MOST EIGHT BITS PER ELEMENT (COMPRESSED)

Array bits per element	m/n	8	14	92
Transmission bits per element	z/n	8	7.923	7.923
Hash functions	k	6	2	1
False positive probability	f	0.0216	0.0177	0.0108

functions can also be changed periodically through a similar mechanism.

For compression issues, arithmetic coding provides a flexible compression mechanism for achieving near-optimal compression performance with low variability. Moreover, arithmetic coding is well understood and has extremely fast implementations for both encoding and decoding. Loosely speaking, for a random m bit string where the bit values are independent and each bit is 0 with probability p and 1 with probability $1-p$, arithmetic coding compresses the string to near $mH(p)$ bits with high probability, with the deviation from the average having a Chernoff-like bound. For more information on arithmetic coding, we refer the reader to [12], [17]. For more precise statements and details regarding the low variability of arithmetic coding, we refer the reader to the Appendix. We note that other compression schemes may also be suitable, including, for example, run-length coding.

Given this compression scheme, we suggest the following approach. Choose a maximum desired uncompressed size m. Then design a compressed Bloom filter using the above theory using a slightly smaller compressed size than desired; for example, if the goal is that the compressed size be z, design the structure so that the compressed size is $0.99z$. This provides room for some variability in compression. Note that the amount of room necessary may depend on m; the 0.99 factor is a rough target that should be subjected to empirical testing, as is done in our examples given below. (The exact amount of room depends on how much overhead the specific compression implementation has, for example; the concentration bounds and bounds for arithmetic coding given in the Appendix can also be used to determine appropriate settings based on the paramaters.) A similar effect may be achieved by slightly overestimating n. If our uncompressed filter is more than half full of zeros, then if we have fewer than expected elements in the set, our filter will tend to have even more zeros than expected and, hence, will compress better. With this design, the compressed filter should be within the desired size range with high probability.

To deal with cases that still do not compress adequately, we suggest using multiple filter types. Each filter type t is associated with an array of size m, a set of hash functions, and a decompression scheme. These types are agreed on ahead of time. A few bits in the header can be used to represent the filter type. If one of the filter types is the standard Bloom filter (no compression), then the set can always be sent appropriately using at least one of the types. In most cases, two types—compressed and uncompressed—would be sufficient.

A. Examples

We test the theoretical framework above by examining a few specific examples of the performance improvements possible using compressed Bloom filters. We consider cases where 8 and 16 bits are used in the compressed Bloom filter for each element; this corresponds to configurations examined in [7], [18]. Also, it is important to note that in these tables, the false positive probability f is given for the Bloom filter in isolation. In an application such as shared Web caching, additional false positives and negatives may arise because changes in the local cache contents may occur between times when the Bloom filters are updated. For a further discussion of this point, see [7], [18].

Suppose we wish to use at most 8 bits per set element in our transmission with a Bloom filter; that is, $z/n = m/n = 8$. Then using the optimal number of hash functions $k = 6$ yields a false positive probability of 0.0216. For $k = 5$, the false positive probability is only 0.0217, so this might be preferable in practice. If we are willing to allow 14 array bits for the uncompressed Bloom filter per set element, then we can reduce the false positive probability by almost 20% to 0.0177 and reduce the number of hash functions to two while keeping the (theoretical) transmitted bits per element z/n below eight, as shown in Table I.

It is also interesting to compare the standard Bloom and the compressed Bloom filter pictorially in this case where $z/n = 8$. In Fig. 1, we show the false positive probability as a function of the number of hash functions k based on the theoretical analysis of Sections II-A and II-B, where we allow k to behave as a continuous variable. Note that, as the theory predicts, the optimized uncompressed filter actually yields the largest false positive probability once we introduce compression. Fig. 2 provides a similar picture for the case where $z/n = 16$.

We tested the compressed Bloom filter via simulation. We repeated the following experiment 100 000 times. A Bloom filter for $n = 10\,000$ elements and $m = 140\,000$ bits was created, with each element being hashed to two positions chosen independently and uniformly at random in the bit array. The resulting array was then compressed using a publicly available arithmetic coding compressor based on the work of Moffat, Neal, and Witten [4], [12].[2] Using $z = mH(p)$ suggests that the compressed size should be near 9904 bytes; to meet the bound of 8 bits per element requires the compressed size not exceed 10 000 bytes. Over the 100 000 trials, we found the average compressed array size to be 9920 bytes, including all overhead; the standard deviation was 11.375 bytes; and the maximum compressed array size was only 9971 bytes, giving us several bytes of room to spare. For larger m and n, we would expect even greater concentration of the compressed size around its mean; for smaller m and n, the variance would be a

[2]We note that this is an adaptive compressor, which bases its prediction of the next bit based on the bits seen thus far. Technically, it is slightly suboptimal for our purposes, since we generally know the probability distribution of the bits ahead of time. In practice, the difference is quite small.

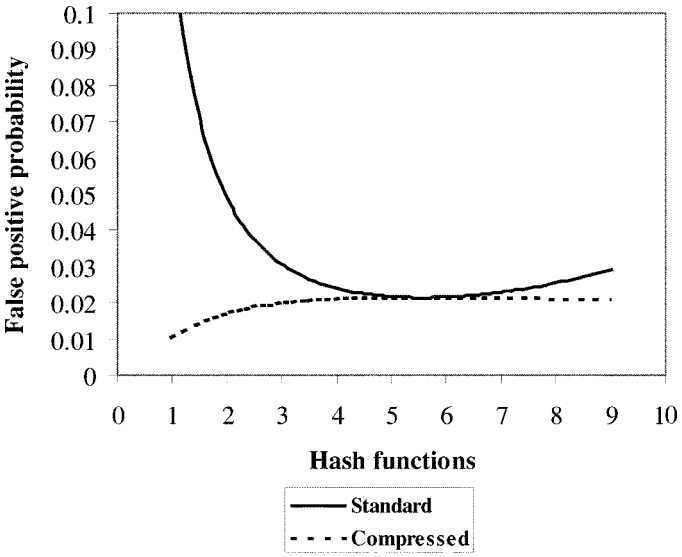

Fig. 1. False positive probability as a function of the number of hash functions for compressed and standard Bloom filters using 8 bits per element.

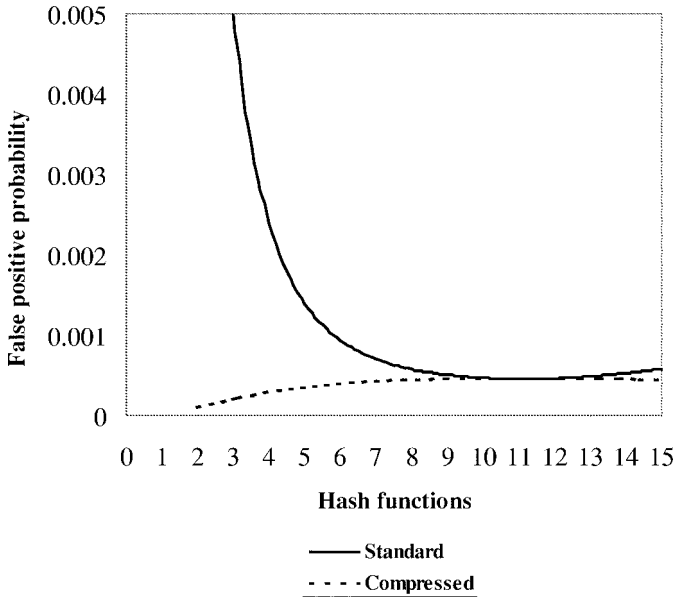

Fig. 2. False positive probability as a function of the number of hash functions for compressed and standard Bloom filters using 16 bits per element.

larger fraction of the compressed size. We believe the example provides good insight into what is achievable in real situations.

Theoretically, we can do even better by using just one hash function, although this greatly increases the number of array bits per element, as seen in Table I.

It is worth noting that if the memory for the Bloom filter array after decompression is a concern, it is often possible to do better by not keeping the Bloom filter in array form. Instead, the array indices where there is a 1 can be kept as a list in sorted order. Checking if an index is 1 can then be accomplished with interpolation search in $O(\log\log n)$ time on average [8]. While this is more than the constant time for an array lookup, it may be suitable for some applications. In the case of one hash function as described above, instead of using $92n$ bits for the uncompressed bit array, only $n\log_2(92n)$ bits could be used for the sorted list; this is much smaller for reasonable values of n.

Similarly, considering the specific case of a Bloom filter where $z/n = m/n = 16$, we would use eleven hash functions to achieve an optimal false positive probability of 0.000 459. As eleven hash functions seems somewhat large, we note that we could reduce the number of hash functions used without applying compression, but using only six hash functions more than doubles f to 0.000 935. Table II summarizes the improvements available using compressed Bloom filters. If we allow 28 array bits per element, our false positive probability falls about 30% while using only four hash functions. If we allow 48 array bits per element, our false positive probability falls over 50% using only three hash functions. We simulated the case with $n = 10\,000$ elements, $m = 480\,000$ bits, and $k = 3$ hash functions using 100 000 trials. The theoretical considerations above suggest the compressed size will be 19 787 bytes. Over our simulation trials, the average compressed array size was 19 805 bytes, including all overhead; the standard deviation was 14.386 bytes and the maximum compressed array size was only 19 865 bytes, well below the 20 000 bytes available.

We have also tested the case where $z/n = m/n = 4$ against using $m/n = 7$, or 7 array bits per element. The results appear in Table III. We expect this case may prove less useful in practical situations because the false positive probability is so high. In this case, using the standard Bloom filter with the optimal three hash functions yields a false positive probability of 0.147; using $m/n = 7$ and one hash function gives a false positive probability of 0.133. Again, we performed 100 000 random experiments with $n = 10\,000$. The largest compressed filter required 4998 bytes, just shy of the 5000 byte limit.

As previously mentioned, we may also consider the optimization problem in another light: We may try to maintain the same false positive ratio while minimizing the transmission size. In Tables IV and V, we offer examples based on this scenario. Our results yield transmission size decreases in the range of roughly 5%–15% for systems of reasonable size. Here again, our simulations bear out our theoretical analysis. For example, using $n = 10\,000$ elements, $m = 126\,000$ bits, and $k = 2$ hash functions over 100 000 trials, we find the average compressed filter required 9493 bytes, closely matching the theoretical prediction. The largest filter over the 100 000 trials required 9539 bytes.

IV. Delta Compression

In the Web cache sharing setting, the proxies periodically broadcast updates to their cache contents. As described in [7], [18], these updates can either be new Bloom filters or representations of the changes between the updated filter and the old filter. The difference, or *delta*, between the updated and old filter can be represented by the exclusive-or of the corresponding bit arrays of size m, which can then be compressed using arithmetic coding as above. For example, one may decide that updates should be broadcast whenever 5% of the underlying array bits have changed; in this case, the compressed size of the delta would be roughly $mH(0.05)$. Hence, one may wish to optimize the array size for a target size of the compressed delta and allow the one-time cost of longer initial messages to establish

TABLE II
AT MOST SIXTEEN BITS PER ELEMENT (COMPRESSED)

Array bits per element	m/n	16	28	48
Transmission bits per element	z/n	16	15.846	15.829
Hash functions	k	11	4	3
False positive probability	f	0.000459	0.000314	0.000222

TABLE III
AT MOST FOUR BITS PER ELEMENT (COMPRESSED)

Array bits per element	m/n	4	7
Transmission bits per element	z/n	4	3.962
Hash functions	k	3	1
False positive probability	f	0.147	0.133

TABLE IV
MAINTAINING A FALSE POSITIVE PROBABILITY AROUND 0.02

Array bits per element	m/n	8	12.6	46
Transmission bits per element	z/n	8	7.582	6.891
Hash functions	k	6	2	1
False positive probability	f	0.0216	0.0216	0.0215

TABLE V
MAINTAINING A FALSE POSITIVE PROBABILITY AROUND 0.000 45

Array bits per element	m/n	16	37.5	93
Transmission bits per element	z/n	16	14.666	13.815
Hash functions	k	11	3	2
False positive probability	f	0.000459	0.000454	0.000453

a base Bloom filter at the beginning. It makes sense to cast this problem as an optimization problem in a manner similar to what we have done previously. As we will show, using compressed Bloom filters in conjunction with delta compression can yield even greater performance gains.

We emphasize that using delta compression may not be suitable for all applications. For example, sending deltas may not be suitable for systems with poor reliability; a missed delta may mean a proxy filter remains improperly synchronized for a long period of time (assuming full filters are sent occasionally to resynchronize). In many cases, however, sending deltas will be preferred.

Suppose that our set S of elements changes over time through insertions and deletions, but the size is fixed at n elements. We send a delta whenever a fraction c of the n elements of the set have changed. We consider the case where our goal is to minimize the false positive probability f while maintaining a specific size for the delta. We again have the power to choose the array size m and the number of hash functions k, given n and the compressed delta size, which we denote here by z. In this setting, we let q be the probability that a bit in the delta is a 1, given that a fraction c of the n elements have changed. Similar to the case for compressed Bloom filters, our constraint is $z = mH(q)$. As before, we let $p = \mathrm{e}^{-kn/m}$ be the probability that a bit in the Bloom filter is 0.

We determine an expression for q in terms of other parameters. A bit will be 1 in the delta in one of two cases. In the first case, the corresponding bit in the Bloom filter was originally a 0 but became a 1 after the elements changed. The probability that the bit was originally 0 is just p; the probability that the cn new elements fail to change that bit to a 1 is $(1 - 1/m)^{cnk} \approx \mathrm{e}^{-cnk/m}$, so (using the asymptotic approximation) the overall probability of this first case is $p(1 - \mathrm{e}^{-cnk/m})$.

In the second case, the corresponding bit in the Bloom filter was originally a 1 but became a 0 after the elements changed. This is equivalent to the previous case. The probability that the bit is 0 at the end is just p, and the probability that the cn deleted elements failed to set that bit to 1 was $(1 - 1/m)^{cnk}$, and the overall probability of this case is also $p(1 - \mathrm{e}^{-cnk/m})$. Hence, $q = 2p(1 - \mathrm{e}^{-cnk/m}) = 2p(1 - p^c)$.

The false positive probability satisfies

$$f = (1-p)^k = (1-p)^{(-\ln p)\cdot(m/n)} = (1-p)^{-(z \ln p/nH(q))}.$$

Since z and n are given, minimizing f is equivalent to minimizing

$$\beta = \mathrm{e}^{-(\ln(p)\cdot\ln(1-p)/H(2p(1-p^c)))}.$$

Unfortunately, we have lost the appealing symmetry of the standard and compressed Bloom filter, making analysis of the above expression unwieldy. The value of β still appears to be minimized as $p \to 1$ for any $c < 1/2$, but a simple formal proof appears challenging.

It is worthwhile to again consider how f behaves in the limiting case as $p \to 1$. Algebraic manipulation yields that in this case $\beta \to (1/2)^{1/2c}$, so f approaches $(0.5)^{z/2cn}$. This result is intuitive under the reasoning that the limiting case corresponds to hashing each element into a large number of bits. The exponent is $z/2cn$ instead of z/n since the updates represent both deletions and insertions of cn elements; half of the bits sent describe the array elements to be deleted.

We present some examples for results in this setting in Tables VI and VII. As before, these tables give the false positive probability f for the Bloom filter in isolation. Also, the tables are based on the analysis above and do not take into

TABLE VI
COMPARING THE STANDARD BLOOM FILTER AND COMPRESSED BLOOM FILTERS WITH DELTA ENCODING; $c = 0.05$

Array bits per element	m/n	8	12	32	13
Transmission bits per element	z/n	1.6713	1.6607	1.6532	1.3124
Transmission bits per element changed	$z/(cn)$	33.426	33.214	33.064	26.248
Hash functions	k	5	3	2	2
False positive probability	f	0.0217	0.0108	0.00367	0.0203

TABLE VII
COMPARING THE STANDARD BLOOM FILTER AND COMPRESSED BLOOM FILTERS WITH DELTA ENCODING; $c = 0.01$

Array bits per element	m/n	8	16	48	13
Transmission bits per element	z/n	0.4624	0.4856	0.4500	0.3430
Transmission bits per element changed	$z/(cn)$	46.24	48.56	45.00	34.30
Hash functions	k	5	3	2	2
False positive probability	f	0.0217	0.0050	0.00167	0.0203

account compression overhead and variability, which tend to have a greater effect when the number of transmitted bits is smaller.

In Table VI, we consider the case where 5% of the elements of S change between updates. A standard Bloom filter using 8 bits per element and five hash functions uses only about 1.67 bits per element when using delta compression. (Another reasonable measure is the number of bits per changed element, instead of the number of bits per element; we include this number in Table VI.) Alternative configurations using more array bits per element and fewer hash functions can achieve the same transmission size while dramatically reducing the false positive probability f. Using four times as much memory (32 bits per element) for the decompressed filter lowers f by a factor of six. The scenario with $m/n = 32$ and $k = 2$ hash functions was tested with simulations. Over 100 000 trials, the average compressed filter required 2090 bytes, closely matching the theoretical prediction of 2066.5 bytes. The maximum size required was 2129 bytes. Alternatively, one can aim for the same false positive ratio while improving compression. As shown in the last column of Table VI, one can achieve the same false positive ratio as the standard Bloom filter while using only about 1.31 bits per element, a reduction of over 20%.

With more frequent updates, so that only 1% of the elements change between updates, the transmission requirements drop below 1/2 of a bit per element for a standard Bloom filter. As shown in Table VII, substantial reductions in the false positive probability or the bits per element can again be achieved.

V. COUNTING BLOOM FILTERS

In [7], the authors also describe an extension to a Bloom filter, where instead of using a bit array the Bloom filter array uses a small number of bits per entry to keep counts. To represent a set, the jth entry is incremented for each hash function h_i and each element x in the set such that $h_i(x) = j$. The *counting Bloom filter* is useful when elements can be deleted from the filter; when an element x is deleted, one can decrement the value at location $h_i(x)$ in the array for each of the k hash functions, i.e., for $1 \leq i \leq k$. We emphasize that these counting Bloom filters are not passed as messages in [7], [18]; they are only used locally.

We note that if one wanted to pass counting Bloom filters as messages, compression would yield substantial gains. The entropy per array entry would be much smaller than the number of bits used per entry, since large counts would be extremely unlikely. Our optimization approach for finding appropriate parameters can be extended to this situation, and arithmetic coding remains highly effective. We expect that similar variations of Bloom filters would benefit from compression as well.

VI. CONCLUSION

We have shown that using compression can improve Bloom filter performance, in the sense that we can achieve a smaller false positive probability as a function of the compressed size over a Bloom filter that does not use compression. More generally, this is an example of a situation where we are using a data structure as a message in a distributed protocol. In this setting, where the transmission size may be important, using compression affects how one should tune the parameters of the data structure. It would be interesting to find other useful examples of data structures that can be tuned effectively in a different manner when being compressed.

Our work suggests several interesting theoretical questions. For example, our analysis depends highly on the assumption that the hash functions used for the Bloom filter behave like completely random functions. It is an open question to determine what sort of performance guarantees are possible using practical hash function families. Also, it is not clear that the Bloom filter is necessarily the best data structure for this problem; perhaps another data structure would allow even better results.

Finally, we have not yet implemented compressed Bloom filters in the context of a full working system for an application such as distributed Web caching. We expect that significant performance improvement will occur even after minor costs such as compression and decompression time are factored in. The interaction of the compressed Bloom filter with a full system may lead to further interesting questions.

APPENDIX
MATHEMATICAL DETAILS

Here, we briefly discuss some of the mathematical issues that we glossed over previously. Specifically, we wish to show that the size of a compressed Bloom filter is very close to $mH(p)$ with high probability. We sketch the argument, omitting the fine detail and focusing on the main points.

We calculated that the expected fraction of 0 bits in a Bloom filter with m bits, k hash functions, and n elements is $p' = (1-1/m)^{nk}$. We proceeded as though the bits in the Bloom filter were independent with probability $p = \mathrm{e}^{-nk/m}$. The difference between p' and p is well known to be very small, as $(1-1/m)^m$ converges quickly to $1/\mathrm{e}$. We will ignore this distinction. The bits of the Bloom filter, however, are also not independent. In fact, as we describe later, for arithmetic coding to perform well, it suffices that the fraction of 0 bits is highly concentrated around its mean. This concentration follows from a standard martingale argument.

Theorem 1: Suppose a Bloom filter is built with k hash functions, n elements, and m bits, using the model of perfectly random hash functions. Let X be the number of 0 bits. Then

$$\Pr[|X - mp| > \epsilon m] < 2\mathrm{e}^{(-\epsilon^2 m^2)/2nk}.$$

Proof: This is a standard application of Azuma's inequality. (See, e.g., [2, Th. 2.1].) Pick an order for the elements to be hashed. Let X_j be a random variable representing the number of 0 bits after j hashes. Then, $X_0, X_1, \ldots, X_{nk} = X$ is a martingale, with $|X_i - X_{i+1}| \leq 1$. The theorem then follows. □

For our arithmetic coding, we suppose that we use an adaptive arithmetic coder that works as follows. There are two counters, C_0 and C_1; C_i is incremented every time the bit value i is seen. Initially, the C_i are set to 1, to avoid the division-by-zero problems discussed below. The encoder and decoder use the model that the probability the next bit is i is to be $C_i/(C_i+C_{1-i})$ to determine how to perform the arithmetic coding. (Thus, initially, when no information is given, the encoder and decoder assume the first bit is equally likely to be a 0 or a 1.)

Recall that for arithmetic coding the total length L of the encoding can be taken to be the logarithm of the inverse of the product of the model probabilities for each bit, plus 1. (See, for example, [9].) In this case, if there are m bits total and x of the bits are 0, regardless of the position of the x 0 bits, the total length L of the encoding satisfies

$$L = \left\lceil \log_2 \frac{(m+1)!}{x!(m-x)!} \right\rceil + 1.$$

We consider the case where $x = \rho m$ for some constant ρ. Simplifying, we have

$$\begin{aligned} L &= \left\lceil \log_2 \frac{(m+1)!}{(\rho m)!((1-\rho)m)!} \right\rceil + 1 \\ &= \log_2 \binom{m}{\rho m} + O(\log m) \\ &= mH(\rho) + O(\log m). \end{aligned}$$

In the above, we used the approximation

$$\binom{m}{\rho m} = 2^{mH(\rho)+O(\log m)}$$

which follows by Stirling's formula for a constant ρ.

Since ρ is with high probability close to p', which is very close to p, the total number of bits used by the encoding is close to $mH(p)$ with high probability.

ACKNOWLEDGMENT

The author would like to thank A. Broder for introducing him to Bloom filters and for helpful discussions.

REFERENCES

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen, "Parallel randomized load balancing," *Random Structures and Algorithms*, vol. 13, no. 2, pp. 159–188, 1998.
[2] N. Alon and J. Spencer, *The Probabilistic Method*. New York: Wiley, 1992.
[3] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, July 1970.
[4] J. Carpinelli, W. Salomonsen, A. Moffat, R. Neal, and I. H. Witten. (1995, Mar.) Source code for arithmetic coding, version 1. [Online]. Available: http://www.cs.mu.oz.au/~alistair/arith_coder/.
[5] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *J. Comput. Syst. Sci.*, vol. 18, pp. 143–154, 1979.
[6] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz, "An architecture for a secure service discovery service," in *Proc. Fifth Annu. Int. Conf. Mobile Computing and Networks (MobiCOM'99)*, Aug. 1999, pp. 24–35.
[7] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *Proc. SIGCOMM'98*, 1998, pp. 254–265.
[8] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd ed. Chatham, NJ: Addison-Wesley, 1991.
[9] P. G. Howard and J. Vitter, "Analysis of arithmetic coding for data compression," *Inform. Process. Manag.*, vol. 28, no. 6, pp. 749–763, 1992.
[10] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," in *Proc. ASPLOS 2000*, pp. 190–201.
[11] M. D. McIlroy, "Development of a spelling list," *IEEE Trans. Commun.*, vol. 30, pp. 91–99, Jan. 1982.
[12] A. Moffat, R. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inform. Syst.*, vol. 16, no. 3, pp. 256–294, July 1998.
[13] M. V. Ramakrishna, "Practical performance of Bloom filters and parallel free-text searching," *Commun. ACM*, vol. 32, no. 10, pp. 1237–1239, Oct. 1989.
[14] A. Rousskov and D. Wessels, "Cache digests," *Computer Netw. ISDN Syst.*, vol. 30, no. 22–23, pp. 2155–2168, 1998.
[15] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakounito, S. T. Kent, and W. T. Strayer, "Hash-based IP traceback," in *Proc. SIGCOMM*, Aug. 2001, pp. 3–14.
[16] D. Wessels. SQUID frequently asked questions. [Online]. Available: http://www.squid-cache.org
[17] I. H. Witten, A. Moffat, and T. Bell, *Managing Gigabytes*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1999.
[18] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol (Extended version)," Computer Sciences Dept., Univ. Wisconsin–Madison, Tech. Rep. 1361, Feb. 1999.

Michael Mitzenmacher (M'01) received the B.A. degree in mathematics and computer science *summa cum laude* from Harvard College, Cambridge, MA, in 1991, the C.A.S. degree in mathematics with highest distinction from Cambridge University, Cambridge, U.K., in 1992, and the Ph.D. degree in computer science from the University of California at Berkeley in 1996.

He was a Research Scientist with the Digital Systems Research Center, Palo Alto, CA, from 1996 to 1998. From 1999 to 2002, he was an Assistant Professor with Harvard University, Cambridge, MA, where he has been an Associate Professor since July 2002. He is the coinventor of ten issued patents. His research interests include design and analysis of algorithms, dynamic processes, load balancing, Web algorithms, compression, error-correcting codes, and computer science education.

Dr. Mitzenmacher was the recipient of an Alfred P. Sloan Research Fellowship in 2000.