Cache-, Hash- and Space-Efficient Bloom Filters

Felix Putze, Peter Sanders, Johannes Singler
{putze,sanders,singler}@ira.uka.de

Fakultät für Informatik, Universität Karlsruhe, Germany

Abstract. A Bloom filter is a very compact data structure that supports approximate membership queries on a set, allowing false positives.
We propose several new variants of Bloom filters and replacements with similar functionality. All of them have a better cache-efficiency and need less hash bits than regular Bloom filters. Some use SIMD functionality, while the others provide an even better space efficiency. As a consequence, we get a more flexible trade-off between false positive rate, space-efficiency, cache-efficiency, hash-efficiency, and computational effort. We analyze the efficiency of Bloom filters and the proposed replacements in detail, in terms of the false positive rate, the number of expected cache-misses, and the number of required hash bits. We also describe and experimentally evaluate the performance of highly-tuned implementations. For many settings, our alternatives perform better than the methods proposed so far.

1 Introduction

The term *Bloom filter* names a data structure that supports membership queries on a set of elements. It was introduced already in 1970 by Burton Bloom [1]. It differs from ordinary dictionary data structures, as the result for a membership query might be true although the element is not actually contained in the set. Since the data structure is randomized by using hash functions, reporting a *false positive* occurs with a certain probability, called the *false positive rate (FPR)*. This impreciseness also makes it impossible to remove an element from a Bloom filter. The advantage of a Bloom filter over the established dictionary structures is space efficiency. A Bloom filter needs only a *constant number* of bits per (prospective) element, while keeping the FPR constant, *independent* from the size of the elements' universe.

The false positives can often be compensated for by recalculating or retransferring data. Bloom filters have applications in the fields of databases, network applications [2] and model checking [4, 5]. The requirements on the Bloom filter and the way of usage differ greatly among these fields of applications.

Paper Outline. In Section 2 we review "standard" Bloom filters which are based on setting k bits in a bit array which are determined by k hash functions. Section 3 introduces and analyzes a family of cache-efficient variants of standard Bloom filters. There are two main ideas here: concentrate the k bits in one (or

only few) cache blocks and precompute random bit patterns in order to save both hash bits and access time. While these Bloom filter variants improve execution time at the cost of slightly increased FPR, the ones presented in Section 4 saves space by engineering practical variants of the theoretically space optimal Bloom filter replacements proposed by Pagh et. al. [11]. The basic idea is a compressed representation of a Bloom filter with $k = 1$. Our main algorithmic contribution is the observation that a technique from information retrieval fits perfectly here: Since the distances between set bits are geometrically distributed, Golomb codes yield almost optimal space [10]. After giving some hints on the implementation in Section 5, we present an experimental evaluation in Section 6. We conclude our paper in Section 7.

2 Standard Bloom Filters with Variants

The original Bloom filter for representing a set of at most n elements consists of a bit vector of length m. Let $c := m/n$ be the bits-per-element rate. Initially, all bits are set to 0. For inserting an element e into the filter, k bits are set in the Bloom filter according to the evaluations of k independent hash functions $h_1(e), \ldots, h_k(e)$. The membership query consists of testing all those bits for the query element. If all bits are set, the element is likely to be contained in the set, otherwise, it is surely not contained.

For a fixed number of contained elements, the FPR is lowest possible if the probability of a bit being set is as close as possible to $\frac{1}{2}$. One can easily show that it is optimal to choose $k = \ln 2 \cdot c = \ln 2 \cdot \frac{m}{n} \approx 0.693\frac{m}{n}$.

The probability that a bit has remained 0 after inserting n elements, is[1]

$$p' := \left(1 - \frac{1}{m}\right)^{kn} \overset{i=kn}{\approx} \lim_{i\to\infty} \left(1 - \frac{kn}{mi}\right)^{i} = \mathrm{e}^{-kn/m}. \tag{1}$$

The false positive rate for a standard Bloom filter (std) is

$$f_{\mathsf{std}}(m,n,k) = (1-p')^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \overset{!}{\approx} \frac{1}{2^k} \tag{2}$$

for the optimal k. The detailed calculation can be found in Mitzenmacher's survey paper [2].

Classification. The original Bloom filter can be termed a *semi-static* data structure, since it does not support deletions. Variants that *do* support deletion are called *dynamic*. The other extreme is a *static* filter where not even additions to the set may occur after construction.

[1] We assume throughout the paper that the hash functions are perfectly random.

Existing Variants for Different Requirements. A variant called *Counting Bloom Filters* [6] allows deletion of elements from the Bloom filter by using (small) counters instead of a single bit at every position. This basic technique is also used by [11] in combination with a space-efficient multiset data structure, to yield an asymptotically space-optimal data structure.

In [7], Mitzenmacher et al. show that we can weaken the prerequisite of independent hash functions. The hash values can also be computed from a linear combination of two hash functions $h_1(e)$ and $h_2(e)$. This trick does not worsen the false positive rates in practice.

3 Blocked Bloom Filters

We will now analyze the cache efficiency of a standard Bloom filter, which we assume to be much larger than the cache. For negative queries, only less than two cache misses are generated, on the average. This is because each bit is set with probability $q = 1/2$, when choosing the optimal k, and the program will return false as soon as an unset bit is found. This cannot be improved much, since at most one cache fault is needed for accessing some randomly specified cell in the data structure.

Standard Bloom filters are *cache-inefficient* since k cache misses are generated by every input operation and (false or true) positive membership query.

In this section, we present a cache-efficient variant called *blocked Bloom filter* (blo). It consists of a sequence of b comparatively small standard Bloom filters (Bloom filter blocks), each of which fits into one cache-line. Nowadays, a common cache line size is 64 bytes = 512 bits. For best performance, those small filters are stored cache-line-aligned. For each potential element, the first hash value selects the Bloom filter block to be used. Additional hash values are then used to set or test bits as usual, but only inside this one block. A blocked Bloom filter therefore only needs one cache miss for every operation. In the setting of an external memory Bloom filter, the idea of blocking was already suggested in [8], but the increase of the FPR was found negligible for the test case there ($k = 5$), and no further analysis was done. The blocked Bloom filter scheme differs from the *partition schemes* mentioned in [7, Section 4.1], where each bit is inserted into a different block.

Let primed identifiers refer to the "local" parameters of the Bloom filter block. On the first glance, blocked Bloom filters should have the same FPR as standard Bloom filters of the same size since the FPR in Equation (2) only depends on k and n/m, since $k = k'$ and since the expected value of n'/m' is n/m. However, we are dealing with small values of m so that the approximation is not perfect. More importantly, n' is a random variable that fluctuates from block to block. Some blocks will be overloaded and others will be underloaded. The net effect is not clear on the first glance. The occupancies of the blocks follow a binomial distribution $B(n, 1/b)$ that can be closely approximated by a Poisson distribution with parameter $n/b = B/c$ since n is usually large, and B/c is a small constant. An advantage of this approximation is that it is independent

of the specific value of n. For the overall FPR of a blocked Bloom filter with local FPR $f_{\mathsf{inner}}(B, i, k)$ we get the following infinite but quickly converging sum:

$$f_{\mathsf{blo}}(B, c, k) := \sum_{i=0}^{\infty} \mathrm{Poisson}_{B/c}(i) \cdot f_{\mathsf{inner}}(B, i, k) \tag{3}$$

For a blocked Bloom filter using the typical value $c = 8$ bits per element, the decline in accuracy is not particularly bad; the FPR is 0.0231 instead of 0.0215 for $B = 512$ bits. By increasing c by one, we can (over-)compensate for that. For larger c, the effect of the non-uniform distribution can be diminished by choosing a smaller k than otherwise optimal. Still, for $c = 20$ and $k = 14$, the FPR almost triples: it rises from 0.0000671 to 0.000194, which might not be acceptable in certain applications. Thus, we have to increase c to 24. The numerically computed results for many values of c are shown in Table 1. These values are impractical for $c > 28$, since more than 50% additional memory must be used to compensate for the blocking. However, for $c < 20$, the additional memory required is only 20%. This can be acceptable, and often even comes with an improvement to the FPR, in the end. For $c > 34$, the blocked Bloom filter with $B = 512$ cannot compensate the FPR any more, for a reasonable number of bits per element.

c	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
c'	6	7	8	9	10	11	12	13	14	16	17	18	20	21	23	25	26	28	30	32	35	38	40	44	48	51	58	64	74	90
+%	20	16	14	12	11	10	9	8	7	14	13	12	17	16	21	25	23	27	30	33	40	46	48	57	65	70	87	100	124	165

Table 1. Increasing the space for a blocked Bloom filter to compensate the FPR (B=512).

Bit Patterns (pat). A cache miss is usually quite costly in terms of execution time. However, the advantage in performance by saving cache misses can still be eaten up if the computation is too complex. For the blocked Bloom filters, we still have to set or test k bits in for every insertion or positive query. On the other hand, modern processors have one or two SIMD units which can handle up to 128 bits in a single instruction. Hence, a complete cache-line can be handled in only two steps.

To benefit from this functionality, we propose to implement blocked Bloom filters using precomputed bit patterns. Instead of setting k bits through the evaluation of k hash functions, a single hash function chooses a precomputed pattern from a table of random k-bit pattern of width B. With this solution, only one small (in terms of bits) hash value is needed, and the operation can be implemented using few SIMD instructions. When transferring the Bloom filter, the table need not be included explicitly in the data, but can be reconstructed using the seed value.

The main disadvantage of the bit pattern approach is that two elements may cause a *table collision* when they are hashed to the same pattern. This leads to an increased FPR. If ℓ is the number of table entries, the collision probability in an n element Bloom filter block is $p_{\text{coll}}(n, \ell) := 1 - \left(1 - \frac{1}{\ell}\right)^n$. Hence we can bound the FPR for one block by

$$f_{\mathsf{pat}}(m, n, k, \ell) \leq p_{\text{coll}}(\ell) + (1 - p_{\text{coll}}(\ell)) f_{\mathsf{std}}(m, n, k)\,. \tag{4}$$

This local FPR can be plugged into Equation (3) to yield the total FPR. Bit patterns work well when on the one hand, the pattern table is small enough to fit into the cache and on the other hand, the table is big enough to ensure that table collisions do not increase the FPR by too much.

Multiplexing Patterns. To refine this idea once more, we can achieve a larger variety of patterns from a single table by bitwise-or-ing x patterns with an average number of k/x set bits. Ignoring rounding problems, dependencies, etc.

$$f_{\mathsf{pat[x]}}(m, n, k, \ell) \approx f_{\mathsf{pat}}(m, xn, k/x, \ell)^x\,. \tag{5}$$

Multi-Blocking. One more variant that helps improving the FPR, is called *multi-blocking*. We allow the query operation to access X Bloom filters blocks, setting or testing k/X bits respectively in each block. (When k is not divisible by X, we set an extra bit in the first $k \bmod X$ blocks.) Multi-blocking performs better that just increasing the block size to XB, since more variety is introduced this way. If we divide the set bits among several blocks, the expected number of 1 bits per block remains the same. However, only k/X bits are considered in each participating block, when accessing an element. Thus, we have to generalize Equation (2):

$$f_{\mathsf{std[X]}}(m, n, k) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn/X}\right)^k \tag{6}$$

We get an estimate for the total FPR of

$$f_{\mathsf{blo[X]}}(B, c, k) := \sum_{i=0}^{\infty} \text{Poisson}_{XB/c}(i) \cdot f_{\mathsf{std[X]}}(B, i/X, k)^X \tag{7}$$

This can be adapted to bit patterns as before. The multiplexing and the multi-blocking factor will be denoted by appending them to either variant, i. e. $\mathsf{blo[X]}$ and $\mathsf{pat[x, X]}$ respectively.

Combinations. Using the formulas presented, all combinations of the variants presented here can be theoretically analyzed. Although the calculations make simplifying assumptions, mainly through disregarding dependencies, they match the experimental results closely, as shown in Figure 1. The differences are very small, and only appear for large c, where random noise in the experimental values comes into play.

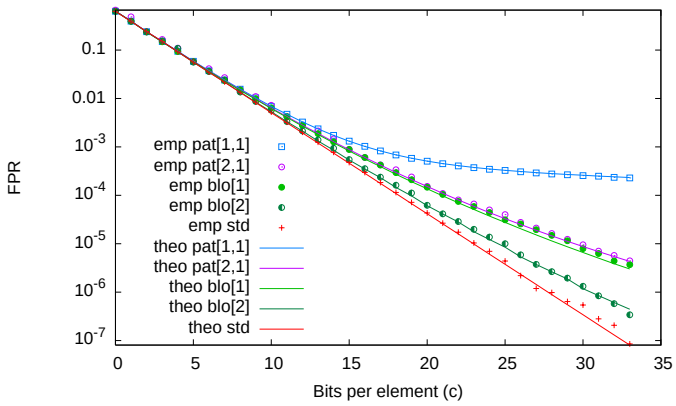

Fig. 1. Comparing the empirical FPR to the theoretically computed one, for some variants. The lines represent the theoretical, the points indicate the experimental results.

Operation	Insert / Positive Query	Negative Query
std	$k \log m$	$2 \log m$
blo[X]	$X \log(m/B) + k \log B$	$\log(m/B) + 2 \log B$
pat[x, X]	$X(\log(m/B) + x \log \ell)$	$\log(m/B) + x \log \ell$
ch	$\log n/f$	$\log n/f$
gcs	$\log n/f$	$\log n/f$

Table 2. Number of hash bits used by the various variants. f is the desired FPR, n the number of elements, and m the available space, B the block size, and ℓ the length of the pattern table. Throughout this paper, $\log x$ stands for $\log_2 x$.

Hash Complexity Besides the cost for memory access, Bloom filters incur a cost for evaluating hash functions. Since the *time* needed for this is very application dependent, we choose to compare the different algorithms based on the number of *hash bits* needed for a filter access. Table 2 summarizes the results.

Exemplary values for $m = 800,000,000$ are shown in Figure 2. The values follow the theoretical computation in Table 2. Obviously, the proposed variants perform better than the standard Bloom filters, for a reasonable choice of c.

4 Space-Efficient Bloom Filter Replacements

In the previous section, we have proposed methods for making Bloom filters produce less cache faults and use less hash bits. The aim was to improve the execution time, while at the same time, sacrificing FPR and/or space efficiency. In this section, we describe Bloom filters with near optimal space consumption that are also cache-efficient. We pay for this with a trade-off between execution time and an additive term in the space requirement.

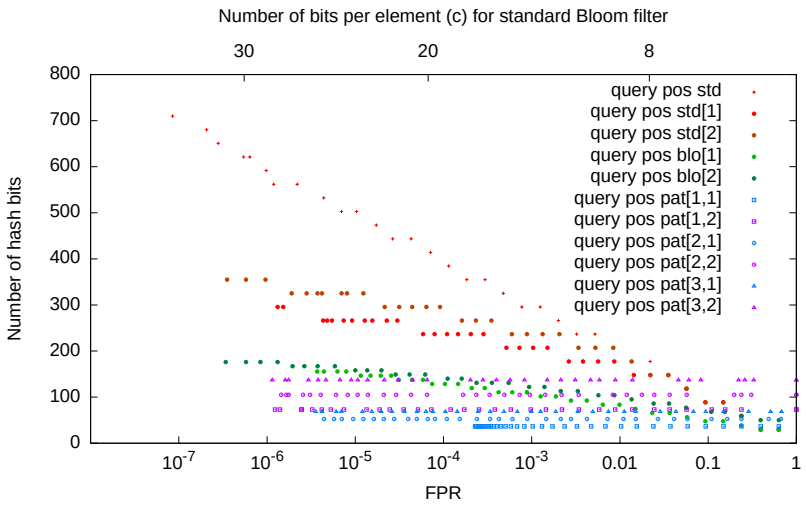

Fig. 2. Number of hash bits used against the FPR.

Our basic solution is also static, i. e. the data structure must be constructed in a preprocessing step, with all elements given, before any query can be posed. At the end of this section we outline how the data structure can be dynamized.

The original Bloom filters are space-efficient, but not space-optimal [11]. When ignoring membership query time, one could just store one hash value in the range $\{1, \ldots, n/f\}$ to obtain an FPR of f. This would cost only $\log \binom{n/f}{n} \approx n \log \frac{e}{f}$ bits instead of $\frac{n}{\ln 2} \log(1/f)$ bits. Hence, a traditional Bloom filter needs about $1/\ln 2 \approx 1.44$ times as much space, even worse by an additive constant when compared to the information-theoretic minimum, $n \log(1/f)$ bits. This amount of extra memory can be saved by sacrificing some access time. Pagh and Pagh [11] use a asymptotically space-optimal hash data structure invented by Cleary [3] for storing just those hash values. Let this approach be termed *CH filter* (ch) here. However, to guarantee expected constant membership query time, a constant number of bits must be spent additionally for each contained element. Those bits comprise a structure that gives some hints to find the desired element more quickly. The more extra bits are provided, the faster the data structures will work. Although the number of bits is independent of n, and more importantly, of the FPR, it eats up most of the savings achieved, for reasonably small values of c. Another point is that a hash data structure should never get close to full, i. e. there must be some maximal load α, which in turn increases memory usage. Summarizing this, access time must be traded off with space efficiency again, but this time with the ability to get arbitrarily close to the theoretical optimum, asymptotically.

Our own solution proposed here is based on an approach used in search engines to store sorted lists of integers [12]. Imagine a simple Bloom filter with $k = 1$, i. e. a hashed bitmap, yielding an FPR of $1/c$. This bitmap can be greatly

compressed, as the 1 bits are sparse. However, differently to [9], we do not use (optimal) arithmetic coding, since this prohibits random access (without unpacking all the data again). Instead, we do not compress the bitmap, but the sorted sequence of hash values in the range $\{0, \ldots, nc\}$ for all the contained elements. These values are uniformly distributed, therefore, the differences between two successive values are geometrically distributed with $p = 1/c$. For a geometric distribution, *Golomb coding* [10, p. 36] is a minimal-redundancy code, i. e. by choosing the right parameter, we lose only at most half a bit per element, compared to the information-theoretic optimum.

However, this compressed sequence still does not allow any random-access, since the Golomb codes have value-dependent sizes. Hence, we have to augment it with a index data structure so we can seek to a place near the desired location quickly. Therefore, we divide the number range of the hash function into parts of equal size I. In addition, for each of these blocks, a bit-accurate pointer to the beginning of the subsequence that contains the corresponding values, is stored. So there is a trade-off once again: For a small search time we want a small I, but large I are good for saving space.

This data structure, termed *Golomb-Compressed Sequence* (gcs) is static, in contrast to the compact-hash approach, i.e., all hash values and thus, all elements in the set must be known beforehand.

Dynamization of gcs We can support insertions by maintaining a small dynamic hash table T_i for recently inserted elements. It suffices if T_i stores the bit positions for the main table. When T_i becomes too big, we empty it by reconstructing the main table. With a bit of caution we can even support deletion using a deletion buffer T_d. This works if both the main table and T_d store multisets of bit positions. This can be done very space efficiently in the main table. We just need to provide a code word for the distance 0. Since this does not significantly increase the lengths of the other code words and since there are only few collisions, the resulting space and time overhead is small.

5 Implementation Aspects

Blocked Bloom filters with bit patterns profit from storing the Bloom filter in *negated* form—a membership query then reduces to testing whether the bitwise-and of the pattern and the negated filter block is zero. Insertion amounts to a bitwise-and of negated pattern and negated filter block.

To scale the hash values to the appropriate range, we use floating-point multiplication and rounding instead division with remainder. Our measurements indicate that this is crucial for performance.

We implemented all algorithms in a library-style way that makes them easy to use in real applications. This includes easy configuration of all tuning parameters, most of them allowed to be changed at runtime. Through generic programming, we are able to plug in arbitrary data types and hash functions, without any runtime overhead. The code can be obtained from the authors.

With all those details described, we can state that everything possible was done to achieve best practical performance for all of the contestants, thus guaranteeing a fair comparison.

6 Experimental Evaluation

We evaluate our implementations using one core of an Intel Xeon 5140 dual-core processor, running at 2.33 GHz with 4 MB of L2-cache using 64 Byte cache lines. They use the automatic vectorization feature of the Intel C++ Compiler[2] to enable SIMD operations without hand-written assembler code. We benchmark the operations

1. insert an element
2. query an element *contained* in the set, returning true
3. query an element *not contained* in the set, returning true or false

The elements are random strings of length 8. They are hashed using one or two variants of Jenkins' hash function [5] with different seeds which output a total of 64 bits. When even more bits are needed, hash values are generated as needed using a linear combination of those two hash values, as proposed in [7]. In each case, the number of elements n is chosen so that the filter data structure has size 95 MB. After inserting n elements, querying for the same set is performed. Additional n elements are queried for in the second step. This made it possible to measure the running times for both positive and negative queries. The cache-line size is 64 bytes, so we chose $B = 512$. For the pattern-based filters, the table size is set to the full 4 MB, resulting in $\ell = 64$K.

To make the comparison fair, we also include variants of the standard Bloom filter that for a given c use a k value just large enough to ensure an FPR at least as good as blo[1] (std[1]) and blo[2] (std[2]) respectively.

Figures 3 and 4 show running times for the positive and negative queries as well as the filter size, for c from 1 to 34. The insertion times are omitted since they are very similar to the times for positive queries.

As stated before, there is not much improvement to expect for negative queries, since their detection is already quite cache-efficient for the original Bloom filter. Also, they do not use many hash bits. For both positive queries and insertions, the blocked Bloom Filter variants outperform the original Bloom filter, in particular for low FPRs, i. e. high reliability. The maximum speedup factor is close to 4, using 32% more memory than the standard variant. However, the speedup is actually smaller than one would expect from the difference in cache misses (see Appendix A). Apparently, the system can hide cache latencies using prefetching.

The normal pattern variants are only slightly faster than the regular blocked Bloom filter. One reason is that we use very cheap hash functions. But there is another cause: When the pattern table occupies all of the cache, almost every

[2] Version 9.1.045

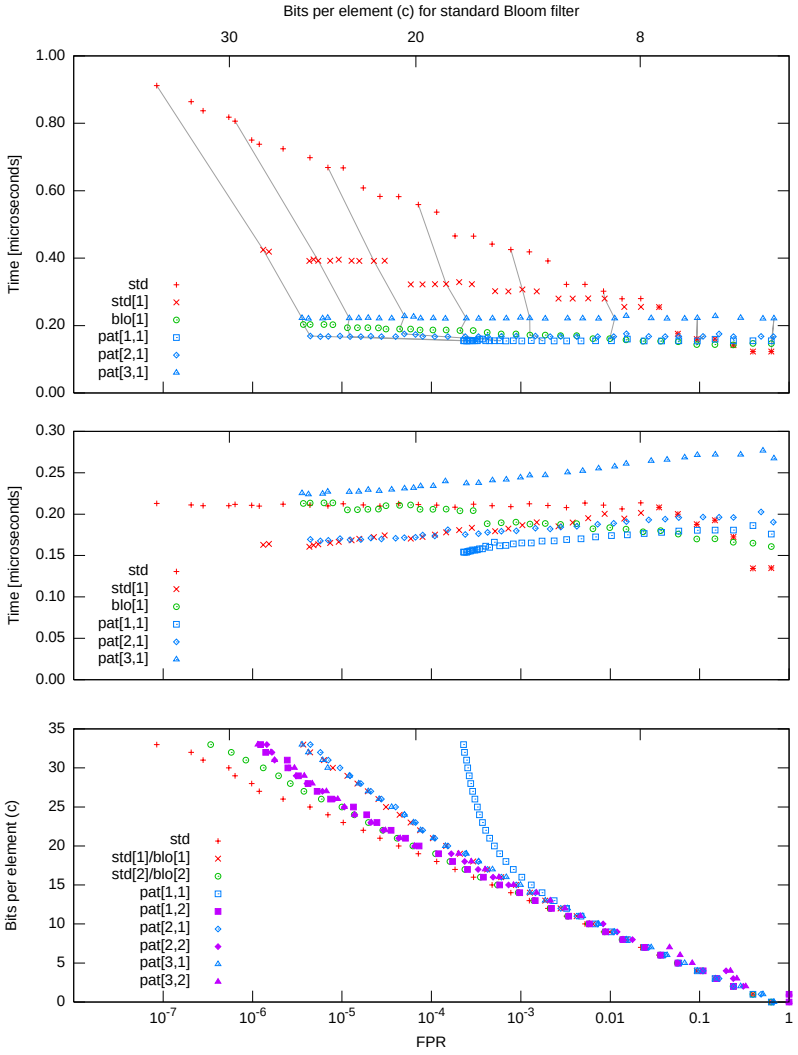

Fig. 3. Execution times for positive (top), negative (middle) queries, and size of filter in bits per contained element (bottom). For comparison, lines connect data points with the same number of bits per element. For readability, only the variants accessing one block are shown here in (top) and (middle), the two-block variants can be found in Figure 4.

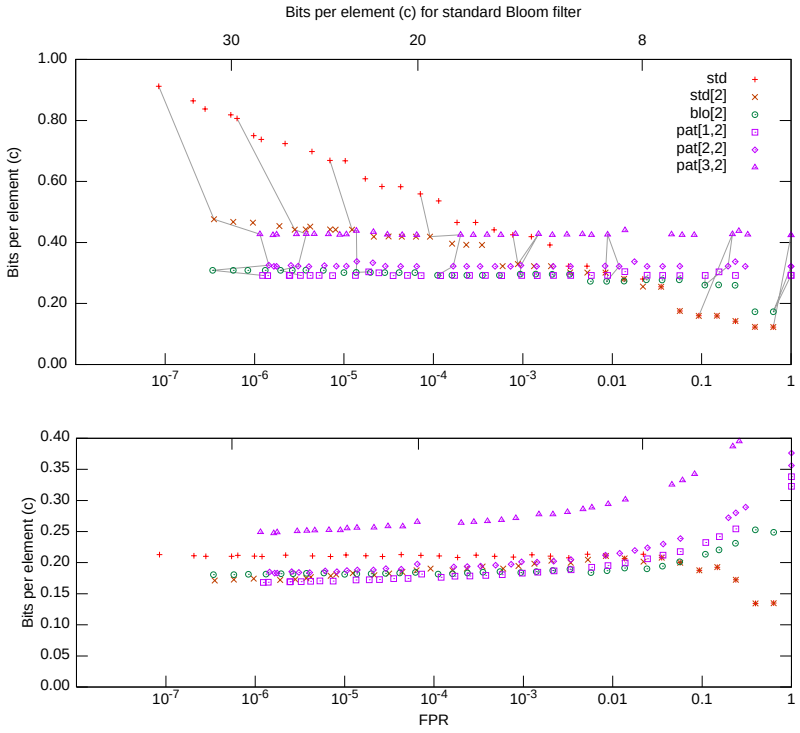

Fig. 4. Execution times for positive (top) and negative (bottom) queries for the variants accessing two blocks.

filter access evicts a pattern from the cache whose next access will cause a cache fault. Reducing the table size (slowly) reduces this problem. The normal pattern variant also does not reach the area of very large FPRs. For $c = 34$, the FPR is limited by the probability of table collisions, about $512/34/64K \approx 2.310^{-4}$. The multiplexing versions are able to overcome this limitation, but need more computational effort.

Regarding the single-blocking variants, for positives and insertions, $\mathsf{pat}[1,1]$ performs best for rather high FPRs, while $\mathsf{pat}[2,1]$ extends this behavior to smaller FPRs, using 2-fold multiplexing.

Regarding the two-blocking variants, for positives and insertions, $\mathsf{pat}[1,2]$ performs best, being up to almost twice as fast as any standard variant, for a low FPRs. It this range, $\mathsf{pat}[1,2]$ can also compete for negative queries. When an even smaller FPR is requested, $\mathsf{blo}[2]$ should be used, which is only marginally slower than $\mathsf{pat}[1,2]$.

We performed a similar test for the space-efficient replacements. The ch data structure used 3 additional bits per entry, while varying the load factor from 0.90 to 0.99.

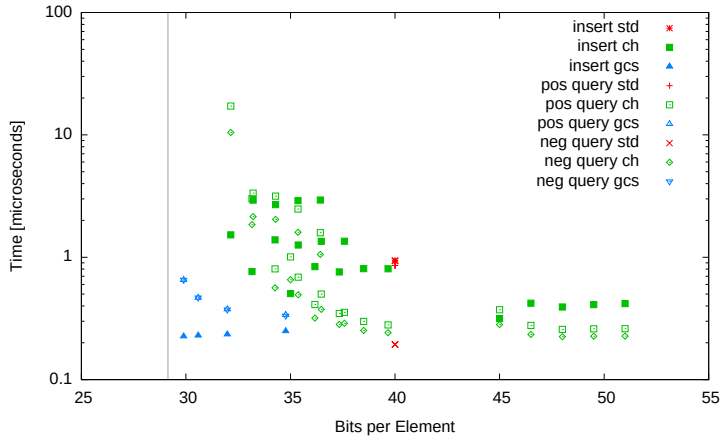

Fig. 5. Execution times of the different operations for many variants and tuning parameters, FPR equivalent to a standard Bloom filter with $c = 40$ and optimal k.

The results are stated in Figure 5, comparing to the standard Bloom filter for $c = 40$, all featuring the same FPR. For this FPR, the lower bound in space for storing the hash values is $-\log 4.5110^{-9} + \log \mathrm{e} = 29.14$ bits per element. The minimum space requirement in this experiment for gcs is in fact 29.14 bits ($I \to \infty$), reaching the optimum, while for ch, it is 30.80 bits ($\alpha \to 1$ and omitting one redundant helper bit per entry). For gcs, the index data structure can be easily and flexibly rebuilt after compact transmission, but for ch, the whole filter must be rebuilt to achieve acceptable execution times.

As we can see, the static gcs implementation provides excellent performance when the memory limitations are tight. If more space is available, Compact hash (ch) gives better query times, but collapses in terms of insertion performance.

7 Conclusion

Which variant or replacement of a Bloom filter works best depends on the application a lot. Standard Bloom filters are still a good choice in many cases. They are particularly efficient for negative queries. Even insertions and positive queries work better than one might think because modern hardware can mitigate cache faults by overlapping multiple memory accesses and because a reduction of k below the "optimal" value brings considerable speedup at moderate increase in space consumption or FPR. Blocked Bloom filters, possibly together with precomputed bit patterns, can mean a significant speedup if insertions and positive

queries are important operations or when hash bits are expensive. Multiplexing and multiblocking Bloom filters become important when a very low FPR is required. Space-efficient Bloom filter replacements are particularly interesting when one wants to reduce communication volume for transferring the filter. Somewhat surprisingly, the price one pays in terms of access time is small or even negative if one uses our implementation based on bucketed Golomb coding. If internal space efficiency is less important than access time and saving communication volume, one could accelerate our implementation further by using Golomb coding only for the communication and by using a faster representation internally.

We believe that, independent of the particular results, our paper is also instructive as a case study in algorithm engineering and its methodology: Modeling both the machine (cache, prefetching, SIMD instructions) and the application (operation mix, difficulty of hashing) are very important here. The paper contains nontrivial components with respect to design, analysis, implementation, experimental evaluation, and algorithm library design. In particular, the analysis is of a "nonstandard" type, i. e. , analysis only seems tractable with simplifying assumptions that are then validated experimentally.

Acknowledgments We would like to thank M. Dietzfelbinger for valuable discussions and hints how to analyze the false positive rate of blocked Bloom filters. Frederik Transier provided a first implementation of Golomb coding.

References

1. B. H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.
2. A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2004.
3. J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, 33(9):828–834, 1984.
4. P. C. Dillinger and P. Manolios. Bloom filters in probabilistic verification. In *FMCAD*, volume 3312 of *LNCS*, pages 367–381, 2004.
5. P. C. Dillinger and P. Manolios. Fast and accurate bitstate verification for SPIN. In *SPIN*, volume 2989 of *LNCS*, pages 57–75, 2004.
6. L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM TON*, 8(3):281–293, 2000.
7. A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *ESA 2006*, volume 4168 of *LNCS*, pages 456–467.
8. U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 25 May 1994.
9. M. Mitzenmacher. Compressed Bloom filters. In *PODC 2001*, pages 144–150.
10. A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer, 2002.
11. A. Pagh, R. Pagh, and S. S. Rao. An optimal Bloom filter replacement. In *SODA 2005*, pages 823–829.
12. P. Sanders and F. Transier. Intersection in integer inverted indices. In *ALENEX 2007*.

Appendix

A Cache-Efficiency Evaluation

	Table Size / Operation					
	64 KB		2048 KB		4096 KB	
Algorithm	insert / pos	neg	insert / pos	neg	insert / pos	neg
std	538601538	111606798	538864877	111633776	538878911	111644012
blo[1]	38564348	38547086	38539297	38522083	38508814	38489184
blo[2]	77068492	38929639	77079012	38936222	77077903	38937805
pat[1,1]	38616001	38567585	54681670	54577097	64018108	63925035
pat[1,2]	77270265	40841892	109155069	55279812	128127287	64866149
pat[2,1]	38656675	38602126	54665523	54575523	64065832	63974708
pat[2,2]	77278236	40868585	109409646	55392837	128379947	65000473
pat[3,1]	38657692	38606045	54601020	54510319	63965985	63891726
pat[3,2]	77203292	40808134	109413361	55396675	128109749	64862561

Table 3. Number of cache misses for various algorithms, operations, and table sizes.

Table 3 lists the number of cache faults that are triggered by executing the experiment. We used a table of size either 64 KB, 2048 KB or 4096 KB. For $c = 20$, we inserted and queried 40,000,000 elements, respectively. blo[1] causes about one cache miss per operation, which is quite accurately reflected by the numbers. For a insertion or positively answered query, the number of cache faults is reduced by a factor of 13.96 compared to std. This is also just as expected, since $k = 14$. However, Figure 3 indicates that for $c = 20$, blo[1] is only about 3 times faster than std. Part of the explanation is that the number of hash bits needed by the two schemes only differs by a factor of about three. However, since the execution time is still dominated by the memory access times, an important part of the explanation seems to be that the compiler schedules memory accesses (or prefetches) already in the loop iteration before the actual use of the data. Thus, cache latency can be hidden behind other operations. This prefetching behavior also explains why there are about 2.9 cache faults per negative query of std although the analysis predicts only two. Apparently, once the query algorithm found a zero bit, one more memory access has already been issued.

For the two-blocking variants, the number of cache misses obviously doubles. When using patterns, the pattern table and the accessed blocks fight for the cache. When the table is as large as the cache, the numbers go up by a factor of 1.7, compared to a table of negligible size. But still, the number of cache misses is far lower than for the standard variants.