Cache Efficient Bloom Filters for Shared Memory Machines

Tim Kaler

tfk@mit.edu
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139

Abstract

Bloom filters are a well known data-structure that supports approximate set membership queries that report no false negatives. Each element in the universe represented by the bloom filter is associated with k random bits in the structure. Traditional bloom filters, therefore, require k non-local memory operations to insert an element or perform a lookups. For very large bloom filters, these k lookups may require k disk seeks. Lookups can be expensive even for moderately sized filters which fit into main memory since k non-local memory accesses may result in L3, L2, and L1 cache misses.

In this paper, we implement a cache-efficient blocked bloom filter that performs insertions and lookups while only accessing a small block of memory. We improve upon the implementation described by [4] by adapting dynamically to unbalanced assignment of elements to memory blocks. The end result is a bloom filter whose superior cache locality allows it to outperform a standard bloom filter on a shared memory machine even when it fits into main memory.

This paper also surveys the design and analysis of three existing types of bloom filters: a standard bloom filter, a blocked bloom filter, and a scalable bloom filter. Ideas from these data structures will allow for the implementation of a cache efficient bloom filter which provides good memory locality. These data structures are used directly by our cache efficient bloom filter to obtain its properties.

1 Introduction

Bloom filters are an efficient data structure for performing approximate set membership queries. A traditional bloom filter maintains an array of m bits to represent a set of size n. It utilizes k hash functions to map each element in some universe to k random bits in its array. An element is inserted by setting its k associated bits to TRUE. A query for approximate set membership is performed for an element by taking the bitwise AND of its k bits. If the element was inserted into the bloom filter, a lookup for that element is guaranteed to be correct. Sometimes, however, an element which is not in the set will be mapped to k bits which have been set due to the insertion of other elements. In this case, a ***false positive*** is reported for that element.

One problem with traditional bloom filters is that they require k random IO operations to perform insertions and approximate membership queries. If a bloom filter is stored on disk, it may be necessary to perform k disk seeks to access k random bits in the filter's array. These random IO operations can also be expensive when the bloom filter fits into main memory.

A modern shared memory machine has a heirarchy of caches. Typically there are three caches of increasing size: L1, L2, and L3. On the modern multicore machine used for the experiments in this paper the L1-cache is 32 KB, the L2-cache is 128 KB, and the L3-cache is 12 MB. Each level of the cache hierarchy is increasingly expensive to access. Even moderately sized bloom filters may exceed 12 MB in space. Accessing k random bits in a bit array much larger than 12 MB may result in up to k expensive L3-cache misses.

This paper explores methods for modifying the standard bloom filter to improve the memory locality of insertions and approximate membership queries. Ideally, each element would be mapped to k bits located on the same cache line (usually 64 bytes) or on the same memory page (usually 4096 bytes). Even coarser memory locality, however, may still help reduce the number of L3-cache misses and improve performance.

Blocked bloom filters, developed by Putze, Sanders, and Singler, provide a potential solution to the poor memory locality of standard bloom filters [4]. A blocked bloom filter is composed of b smaller standard bloom filters of equal size. Each element is mapped to k bits within a single randomly assigned bloom filter. The problem with this approach, however, is that the random assignment of inserted elements to blocks will result in some blocks becoming overloaded. Inserting too many elements into a given block will increase its false probability rate. To compensate, [4] proposes using more space by scaling the size of each block by approximately 15 to 30%.

This paper considers the question: can we efficiently resize each of the b bloom filters dynamically according to its load to save space? The notion of bloom filters which can resize dynamically has been explored previously. ***Scalable bloom filters*** developed by Baquero, Hutchison, and Preguica maintain a predetermined failure probability even when the number of inserted elements is not known in advance [1]. This data structure may be used to implement a ***dynamic blocked bloom filter*** whose composite bloom filters resize dynamically in response to unequal load. This data structure is fairly natural since it is merely a composition of existing work. To the best of the author's knowledge, however, this data structure has not been analyzed in previous work.

Four types of bloom filters have been implemented during the course of this project. The standard bloom filter, the blocked bloom filter, the dynamic (scalable) bloom filter, and the dynamic blocked bloom filter. The first contribution of this paper is a survey of the three bloom filter variants previously described in the literature coupled with some empirical analysis of their performance when implemented. The second contribution is the design and empirical analysis of the dynamic blocked bloom filter which will take advantage of the machinery developed during the course of our survey.

The following is a brief outline of the contents of the paper.

- (Section 2) Design and implementation of the standard bloom filter data structure which requires only 2 independent hash functions, independent of k.
- (Section 3) A scalable bloom filter which can resize itself dynamically at runtime while maintaining the same false positive rate is described and analyzed. Empirical results provide a direct comparison of the false positive rate and memory usage of scalable bloom filters and standard bloom filters. Such a direct comparison appears to have been absent from the original scalable bloom filter paper.
- (Section 4) A blocked bloom filter which is analyzed and empirically tested in two cases: the case in which insertions are equally distributed amongst all blocks, and the case in which insertions are assigned to blocks according to a random hash function.
- (Section 5) A dynamic blocked bloom filter is described and analyzed. The false positive rate, runtime, and memory usage is compared with the standard bloom filter and the blocked bloom filter.
- (Section 6) A brief conclusion which includes a few informal remarks on some related ideas I found interesting, but did not have time to explore in this paper.

2 Standard Bloom Filters

In this section we review the basic theoretical properties of the standard bloom filter for approximate membership queries. We then briefly describe our implementation of the standard bloom filter specifying how to reduce the cost of computing k hash functions and how to guarantee correctness when elements are inserted into the bloom filter concurrently.

Formal description and analysis

Let S be a set of elements that is a subset of some universe U. Let $X_S(n,c,k)$ be a ***standard bloom filter*** utilizing cn bits to perform approximate membership queries on a set of size cn. Each element $e \in U$ can be mapped to k random bits of X_S using k independent hash functions $h_1, \ldots, h_k$. To insert an element e into X_S the bits $h_1(e), \ldots, h_k(e)$ are all set to TRUE. To perform an approximate membership query for the element e the bitwise AND of its k bits in X_S is computed.

The ***false positive rate*** for X_S, f_S, is the probability that X_S reports that an element $e \notin S$ is a member of the set. The parameter k can be optimized to minimize the false positive rate as a function of c. After inserting n elements, the probability that a given bit is empty is $(1 - 1/cn)^{nk}$. This can be rewritten as $((1 - 1/cn)^{cn})^{k/c} \approx e^{-k/c}$. The probability of reporting a false positive for an element $e \notin S$ is equal to the probability of the k random bits assigned to e being set in X_S. This probability is the product $f_S \approx (1 - e^{-k/c})^k$. The approximation for f_S is minimized when $k = c \ln 2 \approx 0.7c$.

We note that for this optimal value of k the probability that any given bit will be TRUE is $(1 - e^{-k/c}) \approx 1/2$ once all elements have been inserted. Therefore, for the optimal values of k and c the false positive rate for X_S will be $f_S \approx 1/2^k$.

Hash functions

The primary implementation concern with the standard bloom filter is the choice of hash functions. If the k bit indicies for a given element $h_1(e), \ldots, h_k(e)$ are expensive to compute then they may dominate the cost of insertions and lookups. The hash functions must have certain properties, however, or else the false positive rate may increase unacceptably. For example, if the k hash functions are only pairwise independent then it has been shown that the false positive rate can increase by a constant factor [3].

It turns out, however, that it is possible to use 2 independent hash functions to simulate $h_1, \ldots, h_k$ while maintaining the same asymptotic false positive rate as when the k hash functions are independent. This result is presented by Adam Kirch, and Michael Mitzenmachert in the paper "Less Hashing, Same Performance: Building a Better Bloom Filter" [3]. Their scheme utilizes two hash functions H_1, H_2, and uses the formula $h_i = H_1 + iH_2$.

Using this technique reduces the problem of computing k random bit indices to that of computing 2 independent hash functions. Our implementation computes the two necessary hash functions H_1, H_2 using the MurmurHash [2] function which provides good performance in practice. We seed each of the two hash functions by utilizing the C library's standard random number generator seeded with the current time.

Concurrent access

This paper will not focus on the concurrent performance of bloom filter variants. It is worth noting, however, that concurrent insertions may introduce the possibility of false negatives in bloom filters. The complexity in implementing a standard bloom filter allowing concurrent inserts arises because most architectures do not support atomic writes to a single bit. Commonly, hardware only guarantees that reads and writes to whole bytes will appear atomic. As elements are being inserted into the bloom filter, multiple elements may attempt to set a bit in the same byte. Suppose two processors attempt to set two different bits to 1 in the same byte b. Each processor will read the byte b into memory, set the appropriate bit, and then store its modified copy of b. If both processors read the byte b before either modifies it, then the first write will be overridden by the second.

In practice, this causes the standard bloom filter to have a non-zero false negative probability! For example $X_S(n = 2^{27}, c = 20, k = 14)$ reports a number of false negatives commonly ranging from 0 to 20 when run on 12 cores. This false negative rate is very small, but it compromises one of the guarantees provided by standard bloom filters. In our implementation, we resolve this race by utilizing the atomic builtin instruction $\_\_sync\_fetch\_and\_or$. It turns out that the use of this atomic instruction does not have a big impact on performance. On simple benchmark with on X_S with 50% of all lookups outside the set S, the nonatomic multicore version runs in ≈ 21.5 seconds, and the atomic multicore versions in ≈ 21.8 seconds.

3 Scalable Bloom Filters

A scalable bloom filter [1] is a bloom filter that can resize itself dynamically while maintaining an upper bound on its false positive rate. We first present the design and analysis of a scalable bloom filter and then provide empirical results using our implementation to demonstrate its false positive rate and space usage.

Design and Analysis

A scalable bloom filter (or dynamic bloom filter) X_D is implemented as a chain of standard bloom filters $X_{S,1}, \ldots X_{S,t}$ whose false positive probabilities decrease geometrically according to a ***tightening ratio*** r. If the false positive probability of $X_{S,1}$ is p, then the false positive probability of the i^{th} filter in the chain is $r^{i-1}p$.

Elements inserted into X_D are added to the last filter in the chain, $X_{S,t}$. When $X_{S,t}$ is unable to insert additional elements without sacrificing its false positive probability a new standard bloom filter $X_{S,t+1}$ is added to the chain. The capacity for newly added bloom filters can be chosen independently of its false positive probability. It is common, however, for the sizes of each filter in the chain to grow exponentially to allow the scalable filter to grow arbitrarily large while guaranteeing a logarithmic chain length.

To perform an approximate membership query on X_D a query is performed on each filter in the chain $X_{S,1}, \ldots X_{S,t}$. The query reports that the element is in the set if any filter in the chain reports that it contains the element and returns FALSE otherwise. The probability of a false positive, therefore, is bounded from above by the probability that any filter in the chain reports a false positive.

$$\begin{aligned} Pr(\textit{False positive in } X_D) &< \sum_{i=1}^{t} r^{i-1}p \\ &< \frac{1}{1-r}p \end{aligned}$$

Tightening false positive rate

In order to tighten the false positive rate, we recall that a standard bloom filter $X_S(n, c, k)$ has a minimal false positive rate when $c = k/\ln 2$ that is equal to $(1-e^{-k/c})^k = (1/2)^k$. If the standard bloom filter has a false positive rate of $p = (1/2)^k$, then a new standard bloom filter $X_S(n', c', k')$ with a false positive rate of $p' = rp$ can be obtained by setting $k' = k + \lg(1/r)$, and $c' = k \ln 2$. Then $p' = (1/2)^{k+\lg(1/r)} = rp$. For our implementations we choose $r = 1/2$ so that $k' = k+1$, and $c' = c+\ln 2$. We utilize floating point representations of k and c to reduce the impact of rounding errors.

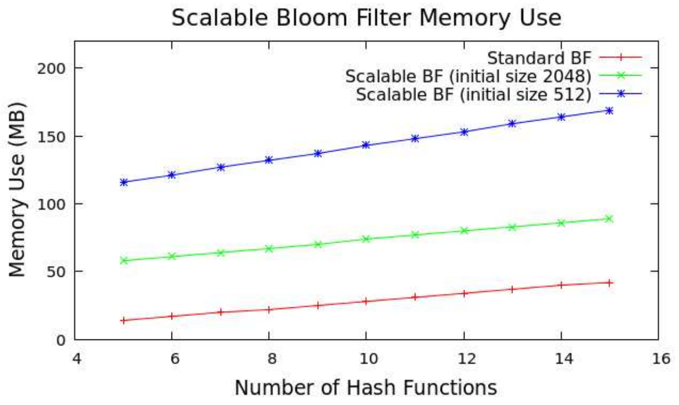


Figure 1: Plot of the memory use (in MB) for scalable and standard bloom filters as a function of the number of hash functions utilized. Results for the scalable bloom filter are shown for two initial set sizes: 512 elements and 2048 elements.

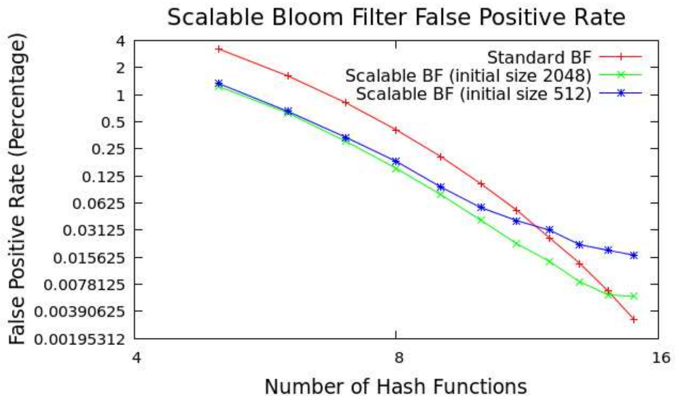


Figure 2: Plot of the false positive rate for scalable and standard bloom filters as a function of the number of hash functions utilized. The x-axis is on a log (base 2) scale. Results for the scalable bloom filter are shown for two initial set sizes: 512 elements and 2048 elements.

Initializing a scalable bloom filter

Suppose that we wish to initialize a scalable bloom filter that maintains the same false positive probability of a standard bloom filter that knows its size in advance: $X_S(n, c, k)$. Suppose that the false positive probability of X_S is q. The bound on the scalable bloom filter's false positive rate shows that to guarantee a false positive rate of q the false positive probability of the first bloom filter in the chain must be $p = (1 - r)q$. As we saw when we tightened the false probability rate of each filter in the chain, this requires us to increase k by $\lg(1/(1 - r))$ and the number of bits per element by $\lg(1/(1 - r))/0.7$. For example, if $r = 1/2$ this, then we increase k by 2 and the number of bits per element by $2/0.7 \approx 2.85$. The first bloom filter in the chain, therefore, will be $X_{S,1}(n', c + 2.85, k + 2)$ where n' is an initial estimate of the size of the set.

Implementation, and empirical results

I ran an experiment to compare the false positive rate and memory usage of the standard and scalable bloom filters. A total of $n = 2^{25}$ random elements (pseudorandom integers) were inserted into each bloom filter. Then $2n$ lookups were performed n of which were guaranteed to have been inserted into the set. An exact set data structure containing all inserted elements was used to compute the exact number of false positives. The standard bloom filter was initialized assuming a set of size n, but the scalable bloom filters used smaller intial set sizes of 512 and 2048. The tightening ratio for the scalable bloom filters was chosen to be $r = 1/2$ and the capacity of $X_{S,i+1}$ was set to be double the capacity of $X_{S,i}$, bounding the length of the chain by $\lg n$.

The result of the experiment measuring memory usage is displayed in Figure 1. Note that the initial size of the bloom filter can have a big impact on memory usage. The scalable bloom filter intially sized to contain 512 elements uses roughly double the space of the scalable bloom filter initially sized to contain 2048 elements. The reason for this increased space usage is that each bloom filter in the chain requires additional bits for each inserted element to guarantee a tightened false positive rate.

An additional phenomenon we note is that the initial size of the scalable bloom filter appears to have an impact on its false positive rates in practice. A comparison of the scalable bloom filters initialized with 512 and 2048 elements with the standard bloom filter in Figure 2 reveals that the filter with initial capacity of 2048 elements matches the false positive rate of the standard bloom fitler for larger values of k. The false positive rate of the filter with initial capacity 512 grows larger than that of the standard bloom filter when more than 12 hash functions are used. The filter with initial capacity 2048, however, has a false positive rate that is lower than the standard bloom filter until 15 hash functions are used. It is, of course, expected that the inital capacity would have some effect on the false positive rate because the false positive rate of a scalable bloom filter experiencing t resize operations will be the sum of t terms in a geometric series converging to its desired probability. The magnitude of the effect, however, is somewhat surprising. The original work on scalable bloom filters in [1] did not provide a direct empirical comparison of the false positive rates of scalable and standard bloom filters. For this reason, it is unclear whether this phenomenon is due to a subtle implementation bug (perhaps a rounding error) or has some other cause (perhaps related to the imprecision of various approximations).

4 Blocked Bloom Filters

In this section we describe the blocked bloom filter as described in [4] and analyze its real world performance when its blocks are evenly and unevenly loaded.

Design and analysis

Suppose we have a blocked bloom filter $X_B(n, c, k, b)$ where b is the number of blocks. Let n be the total number of elements inserted into X_B. As with the standard bloom filter, we will store c bits for each element n. Each block is a standard bloom filter of capacity n/b containing cn/b bits. To perform insertions and lookups a hash function $s : U \to \{1, \ldots, b\}$ is used to assign elements from the universe U to one of b bloom filter blocks. The insertion or lookup for that element is then performed on its assigned bloom filter. Our implementation utilizes the MurmurHash function (with a distinct seed) to perform this sharding.

We now will estimate the false probability rate of this blocked bloom filter. After all elements have been inserted the false positive probability of each bloom filter block will be fixed. Let f_{B_i} be the false positive rate of the i^{th} bloom filter block. Consider an element e which was not inserted into X_B. To lookup e we map e to one of b bloom filter blocks, B_i. We then perform a regular bloom filter lookup for e in block B_i. The false positive probability for the i^{th} bloom filter is f_{B_i}. Therefore, we have that $Pr(\textit{false positive on } e | e \to B_i) = f_{B_i}$. Using the chain rule of probability we can compute the probability of a false positive when looking up e in X_B.

$$Pr(\textit{false positive on } e) = \sum_{i=1}^{b} \frac{f_{B_i}}{b}$$

Perfectly balanced case

The false positive rate of a blocked bloom filter depends on the distribution of inserted elements among its b blocks. In general, the more unevenly elements are distributed among its blocks the larger the false positive probability. The best case performance of the blocked bloom filter, therefore, can be ascertained by testing its performance on an insertion set which shards evenly into its b blocks.

In this ideal case, we can analyze the theoretical false positive rate of the blocked bloom filter. Each block is a standard bloom filter with cn/b bits. By assumption, each block has received n/b insertions. Therefore, the

Bloom Filter	Cache References	Runtime	False Positive Rate
1. Standard BF	45.257 M/sec	172.53s	0.014%
2. Blocked BF Balanced	30.250 M/sec	154.97s	0.013%
3. Blocked BF Unbalanced	32.695 M/sec	163.81s	0.021%

Figure 3: Comparison of the cache performance and runtime of the several variants of the standard bloom filter and the blocked bloom filter under different sharding distributions. The cache misses statistic was gathered using the *perf stat* command. These experiments were run using parameters $n = 2^{27}$ elements, $c = 20$, $k = 13$. n insertions were performed followed by $2n$ lookups (half of which were contained in the set).

false positive probability for any block i is $f_{B_i} = (1 - e^{-k/c})^k$. Since we know the false positive rate of each of the b bloom filters we can compute the probability of reporting a false positive for an element for an element $e \notin S$.

$$\begin{aligned} Pr(\textit{false positive on } e) &= \sum_{i=1}^{b} \frac{f_{B_i}}{b} \\ &= (1 - e^{-k/c})^k \end{aligned}$$

Indeed, we observe that in practice blocked bloom filters have false positive probabilities that are asymptotically the same as the standard bloom filter with the same parameters c, k when their blocks are balanced.

Empirical evaluation

I performed an experiment to measure the relative cache efficiency of the standard bloom filter and the blocked bloom filter under balanced and unbalanced distributions of insertions to blocks. The blocked bloom filters store a fixed number of elements in each block. Each block contains 2944 elements using approximately 6835 bytes of space. The runtime, total number of cache references, and false positive rates are reported in Figure 3.

The total number of cache references for the blocked bloom filter is approximately 30% lower than for the standard bloom filter under both balanced and unbalanced insertion distributions. This translates into a roughly 10% improvement in overall runtime. The false positive rate for the unbalanced blocked bloom filter, however, is larger than the false positive rate for the standard bloom filter by a factor of 1.5. The blocked bloom filter with balanced insertions, however, has a false positive rate that is actually slightly lower than that of the standard bloom filter.

Unbalanced case

The unbalanced assignment of inserted elements to bloom filter blocks has an impact on the false positive rate for blocked bloom filters as seen in Figure 3. If each inserted elements is assigned a random block then we expect the sizes of the blocks to follow a binomial distribution. Experimentally, when inserting 2^{25} elemenst into 655360 bins the average is 51 element, but the minimum is 20 and the maximum is 90.

The remainder of this paper will discuss how "scalable bloom filters" can be used to allow each bloom filter block in X_B to grow dynamically when it is overloaded.

5 Dynamic Blocked Bloom Filter

The blocked bloom filter previously described was composed of b standard bloom filters whose sizes were determined statically based on their expected sizes. The ***dynamic blocked bloom filter*** is a blocked bloom filter in which the size of each block is grows dynamically as the number of elements inserted increases. This allows the dynamic blocked bloom filter to maintain a low failure probability without needing to scale the size of every block by a fixed percentage. In practice, the failure probabilities provided by the dynamic blocked bloom filter match those of the standard bloom filter fairly closely for values of k between 5 and 15.

Design and analysis

The dynamic blocked bloom filter is composed of b blocks. Each block is a scalable bloom filter with initial capacity n/b. The tightening ratio used for the scalable bloom filters was chosen to be $r = 1/2$ for simplicity. Based on the analysis in Section 3 we match the false positive rate of a standard bloom filter with parameters c, k by configuring each scalable bloom filter to initially use $k' = k + 2$ hash functions and $c' = c + 2/0.7$ bits per elements. As seen in Figure 4 this additional space requirement has an impact on the space efficiency of dynamic blocked bloom filters for small values of k. For $k > 8$, however, the total space used (including dynamic resizing) is less than 20% of the space used by a standard bloom filter.

When a scalable bloom filter reaches its capacity it resizes itself by adding an additional bloom filter to its chain with a tighter false probability rate. In Section 3 we increased the size of each filter in the chain exponentially so that the length of the chain would be logarithmic in the number of inserted elements. Exponential growth is not required to guarantee a logarithmic chain

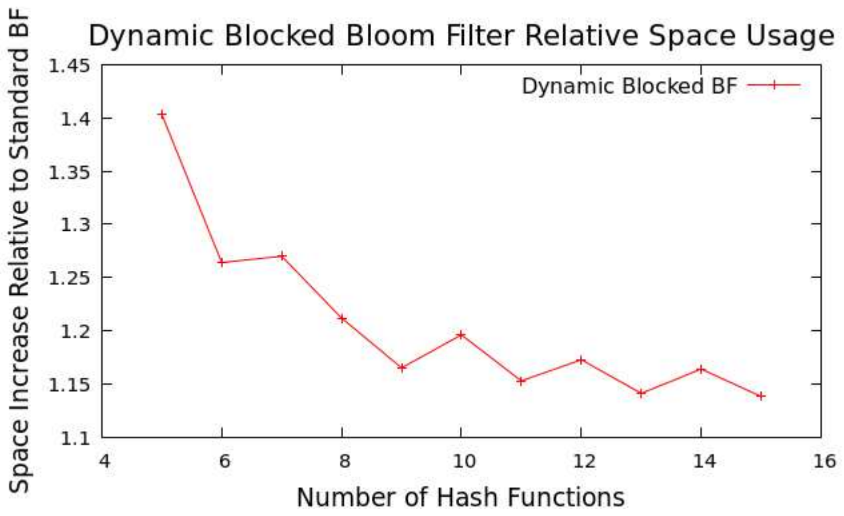


Figure 4: Plot of the relative space usage of the dynamic blocked bloom filter in comparison to the standard bloom filter as a function of the number of hash functions used. A point (x, y) on the plot is implies that for $k = x$, the dynamic blocked bloom filter uses y times as much space as the standard bloom filter.

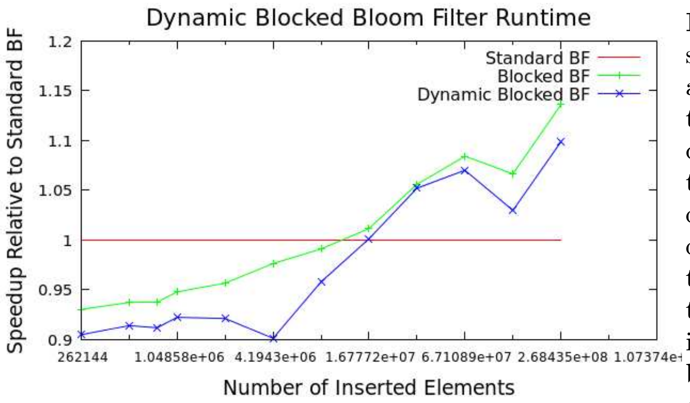
length for scalable bloom filter blocks, however, because with high probability no scalable bloom filter block will grow to be larger than $\Theta(n \log(n)/b)$. To reduce space usage, scalable bloom filter blocks grow by a fixed amount defined to be a fraction of their initial capacity.

Performance results

I ran experiments to compare the runtime and false positive rate of the dynamic blocked bloom filter against the blocked and standard bloom filters.

Figure 5 contains two plots of the relative speedup achieved over the standard bloom filter by the blocked and dynamic blocked bloom filters. When the bloom filter size is smaller than 8 MB the blocked bloom filters are slower than the standard bloom filter. The relative performance between the standard and blocked bloom filters is nearly constant when the bloom filter size is less than 8 MB. However, as the bloom filter size grows to be approximately 16-32 MB the performance gap between the standard and blocked bloom filters closes rapidly. After the bloom filter exceeds 32MB in size the blocked bloom filters perform better than the standard bloom filter.

Why does the relative performance change so rapidly as the set size enters the 16-32MB range? A likely explanation is that it is within this range that the standard bloom filter begins to incur expensive L3 cache misses that the more cache efficient blocked filters avoid. The L3 cache on the experimental



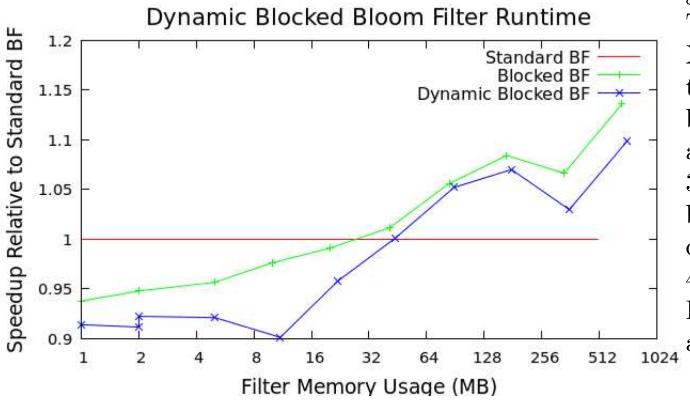


Figure 5: Two plots showing the speedup achieved by the static and dynamic blocked bloom filter when compared to the standard bloom filter. The x-axis of the first plot shows the number of elements inserted. The x-axis of the second plot shows the total space in MB used by the filter. The benchmark was run using $k = 15$ hash functions. These benchmarks were run on a single core of a multicore machine. The machine used was an Intel Xeon X5650 with 49GB DRAM, two 12-MB L3-caches each shared between 6 cores, and private L2- and L1-caches with 128 KB and 32 KB, respectively. Since the benchmark was run on a single core the program had access to 4 GB of DRAM, a single 12-MB L3-cache, a 128 KB L2-cache, and a 32 KB L1-cache.

machine is approximately 12 MB which is quite close to the point at which the relative performance between the standard and blocked filters begins to change.

Figure 6 demonstrates that the dynamic blocked bloom filter has a lower false positive rate than the static blocked bloom filter and can match the false positive rate of the standard bloom filter for values of k between 5 and 15.

6 Conclusion

This paper has surveyed and implemented three known bloom filter variants: the standard bloom filter, the scalable bloom filter, and the blocked bloom filter. In addition, we presented a fourth bloom filter called the dynamic blocked bloom filter which utilizes ideas from blocked and scalable bloom filters to obtain a cache efficient bloom filter which maintains tighter failure probabilities. Improving cache locality appears to be worthwhile for bloom

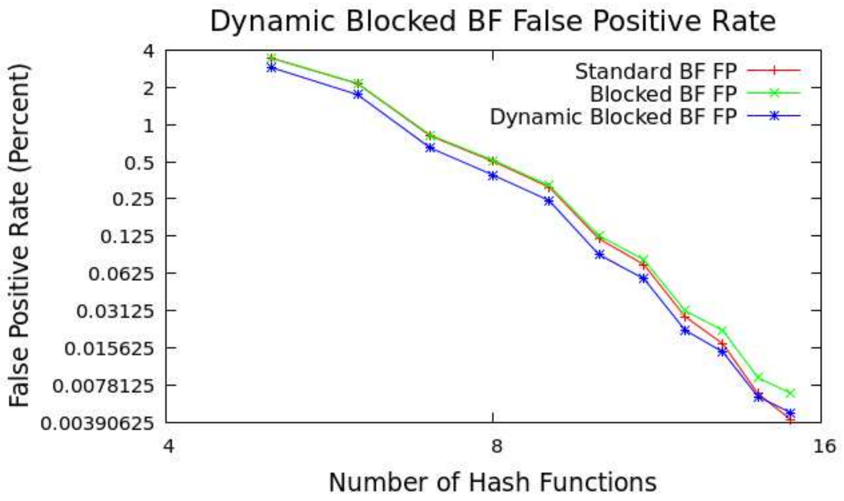


Figure 6: Plot of the false positive rate of the dynamic blocked bloom filter, the blocked bloom filter and the standard bloom filter as a function of the number of hash functions used. The x-axis is on a log (base 2) scale. The false positive rate was calculated by inserting 2^{24} random elements into each set and then performing 2^{25} lookups. Half of the lookups were guaranteed to be positive. False positives were detected using an exact set data structure containing all inserted elements.

filters even when they fit into main memory. Performance improvements were observed for cache efficient bloom filter variants when filters were larger than the experimental machine's L3 cache. The source code for this project is on bitbucket (`https://bitbucket.org/tfk/bloom-project`).

Future exploration

The following are extensions and ideas, described informally, which did not make their way into this paper. It may be interesting the explore these ideas more fully in future work.

Overflow table An alternative approach to dynamically resizing the bloom filter blocks is to use an additional bloom filter to contain all elements which have "overflowed." If an inserted element cannot be added to a bloom filter block without causing that block to exceed its capacity, then it is inserted into the overflow bloom filter. This method can be applied recursively so that the overflow bloom filter could be, itself, a cache efficient bloom filter that handles overflow. Not all lookups would be required to query the overflow filter: only those which queried a block that was at capacity. Still, since some lookups may need to query the overflow table it is necessary to tighten the false probability rate: requiring more bits per inserted element. Fortunately, however, the size of the overflow table need only be a fraction of the size of the original table since most inserted elements will not overflow.

External memory setting The performance benefits of using blocked bloom filters are magnified in the external memory setting where the cost of a random IO is quite large. It may be interesting to investigate ways to improve the performance of blocked bloom filters in the external memory setting. For example, for dynamic blocked bloom filters it is likely worthwhile to ensure that each dynamic block is contained in a contiguous portion of the disk. This could be accomplished by copying the entire dynamic block to a new portion fo the disk whenever it grows.

References

[1] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, Mar. 2007.

[2] A. Appleby. Murmurhash2, 2011.

[3] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, Sept. 2008.

[4] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics*, 14:4:4.4–4:4.18, Jan. 2010.