Hierarchical Bloom Filters: A Page Fault Reducing Bloom Filter

Adam Zawierucha (adz2)

Rice University
zawie@rice.edu

Abstract

Bloom filters [1] are used ubiquitously due to their speed and memory efficiency in theory and in practice. However, the default implementation of sufficiently large bloom filters sets and reads bits appearing in different physical pages of memory, which results in an increase in page faults. In general, increase memory usage increases page faults which in turn slows down programs [4]. In this paper we propose a bloom filter implementation that guarantees one page access per insert or query, minimizing page faults. The implementation we propose implements the abstract bloom filter operations on a collection of baby bloom filters with a size equal to the system's page size. We will show theoretically and empirically that this implementation is expected to be faster than the default implementation.

Keywords: probabilistic data structures, bloom filters, memory hierarchy, implementation, page fault analysis

1 Descripton

In this section I will describe the proposed hierarchial blooom filter implementaiton in detail.

Our hierarchial bloom filter will be a collection of bloom filters of the computer systems page size (4096 bytes). We will have as many bloom filters as it takes to occupy M bits. M sohuld be $10N$ where N is the number of expected keys to be in the bloom filter. We will generate 8 independent hash functions. 1 hash function will be used to select the sub-bloom filter; the other 7 will be used to set bits in the baby bloom filter as the standard implementation.

The idea behind this is that we minimize page faults, as we only need to access one page of memory to set bits instead of up to 7.

The numbers selected will be justified in the *Theoretical Justification* section.

2 Literature Survey

In my literature survey, I discovered a similar approach except making bloom filters hierarchial. Tim Kaler's proposed cache-efficient bloom filters [3] makes sub-bloom filters of size cache-size; thus, their idea is very similar to mine except they do it on a smaller unit of memory.

Evgeni Krimer and Mattan Erez used a power-of-two choice principle within blocked-bloom filters to decrease the false positive rate. Instead of simply selecting one block to write into, they choose multiple. [2].

3 Performance Theoretical Justification

Asymptotically, we expect no difference between the variantions. However, real life computer systems have memory hierarchy; in essence, certain parts of memory are faster to access than others. One unit of memory is a PAGE – typically a 4096 contiguous block of physical memory. The standard variation sets bits accross a virtually contiguous block of M bits of memory. However, physically this is subdivided into pages. Thus, for sufficiently large M it becomes increasingly likely that the bits set and read by the standard implementation will origin from distinct physical pages. In fact, for a bloom filter of size 100 pages and $k = 7$, we expect to hit 6.8 pages. Thus, we can make up to 7 page faults.

Proof. Let M denote the number of bits in the bloom filter. Let P denote the number of bits in a page. Let k denote the number of hash functions used.

Let A be a random variable denoting the number of pages accessed. Let A_i be an indicator variable indicting whether or not page i is accessed.

$$E(A_i) = P(\text{page } i \text{ is not accessed by any of the } k \text{ hash functions})$$

$$E(A_i) = 1 - (P(\text{a hash function does not access page } i))^k$$

$$E(A_i) = 1 - (1 - P(\text{a hash function hits page } i))^k$$

$$E(A_i) = 1 - (1 - \frac{1}{M/P})^k$$

$$E(A_i) = 1 - (1 - \frac{P}{M})^k$$

By the linearity of expectation:

$$E(A) = \Sigma_0^{M/P}(1 - (1 - \frac{P}{M})^k)$$

$$E(A) = \frac{M}{P}(1 - (1 - \frac{P}{M})^k)$$

When we set $k = 7$, $\frac{M}{P} = 100$ (in other words, we use 100 pages of memory), we see that:

$$E(A) = 6.8$$

□

Thus, the standard implementation expects to see in the vincity of k page faults for sufficiently large M.

Our hierarchial bloom filter gaurentees 1 page fault. This is trivially true as we contructed a hash function which decided which page we will write all our bits into. Thus, our implementation will experience as many or less page faults! Thus, we expect higher throughput as the operating systems does not have to load and store pages of memory as frequently.

4 False Positive Theoeretical Justification

In this section, we will justify theoretically why our implementaiton will have a similar false positive rate.

4.1 Standard

The anticipated false postive rate of the standard implementation of bloom filter is:

$$f_p = (1 - (1 - \frac{1}{m})^{nk})^k \tag{1}$$

4.2 Hierarchical

$$f_p = (1 - (1 - \frac{1}{m})^{nkl})^{kl} \tag{2}$$

Where l is the number of sub-bloom filters to access. And k is the number of hash functions to use per sub-bloom filter.

For similar reasons, the optimal value for $kl = 7$. Thus, we select $l = 1$ and $k = 7$ to minimize page access while minimizing the false positive rate. This assumes $m = 10n$.

5 Experiments

I run three experiments on my data structure.

5.1 Measure False Positive Rate

For this experiment, we seek to measure how the false positive rate of our Hierarchical bloom filter compares to the standard bloom filter. Our theoeritcal anaylsis shows that these should be equivalent.

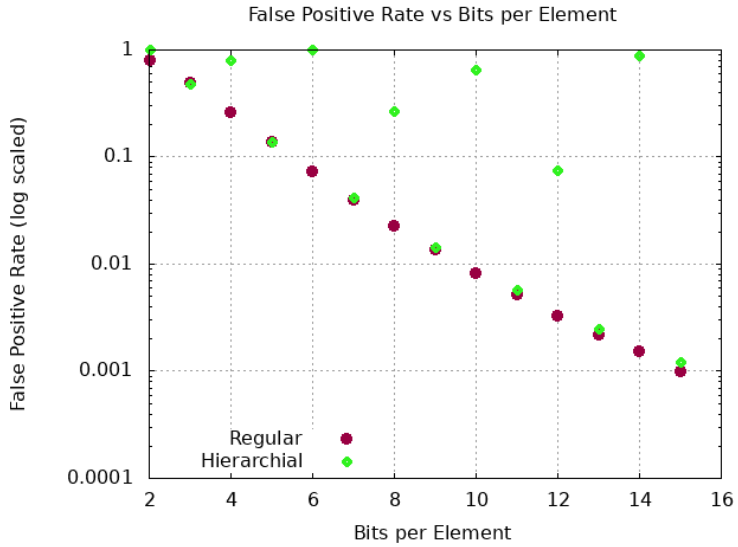
Hypothesis: *The hierarchial bloom filter will have an identical false positive rate.*

5.1.1 Experimental Settings

The experiment will run as follows. First, generate N random keys to insert and N random keys to query by that are all distinct from the insertion keys. We choose $N = 2,000,000$. Then, for each bloom filter variant, generate bloom filters of various sizes: a bloom filter with $2N$ bits, $3N$ bits, ... to $15N$ bits. Generate the bloom filter with the parameters that theoertically minimize false positive rate; i.e $k = 7$, $l = 1$. Then, for each sized bloom filter, insert our N keys for insertion and query N false keys and compute what precentage of them are falsely accepted. This measurement is the false positive rate of the variant for a certain bits per element measurement. Repeat this experiment 20 times and take the average. (The plots I generated in results only do it once, for now.) This gives us a good gauge of how false positive rate behaves for each variant as we provide more memory.

Ideally, we want to see the Hierarchical bloom filter having the same false positive rate as the standard bloom filter.

5.1.2 Results



We can see interesting results here. The hierarchial bloom filter has substantially worse false positive rates sometimes. You can see sometimes the false positive bad becomes extraordinarily bad. These upticks are likely due to bad selections of seed for the choke-point hash function. Thus, this leads to a lot of collisions and therefore a high false positive rate.

For the filnal paper I will run the experiment multiple times and mean to get a more meaningful result.

Nonetheless, this demonstrates a flaw in my proposed data structure which I will discuss more in my final paper. It is interesting that the theoeretical analysis predicts that they will have the same bound, but in practice we see worse false positive rate.

5.2 Measure Throughput for small m

For this experiment, we seek to validate two things

1. The run time is constant with respect to the size of the bloom filter
2. The hierarchical bloom filter has similar speeds to the standard bloom filter for low page sizes.

Hypothesis: *The hierarchial bloom filter will work the same 1 page size, and slowly become better but platue when the page count becomes 7.*

5.2.1 Experimental Settings

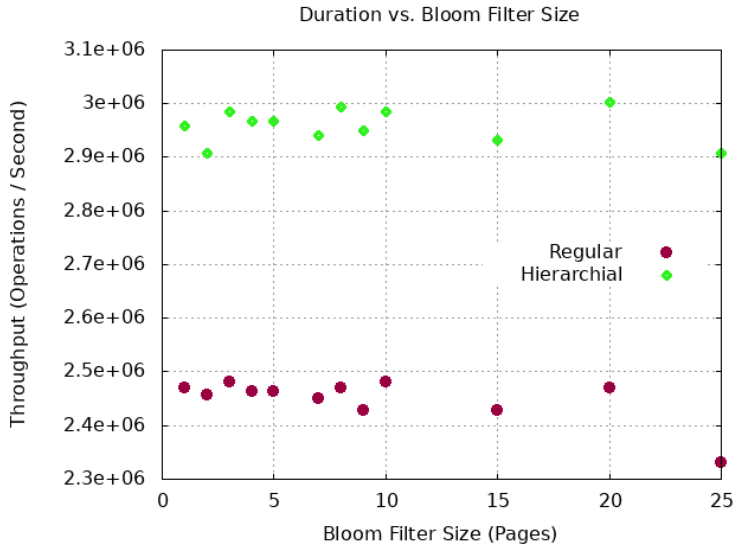
The experiment will run as follows. First, generate $N = 10,000,000$ random keys.

Then, we will run the following experiment on the two variants and plot the results. Use the optimal theoeretical configuraiton for each bloom filter (i.e, $k = 7$, $l = 1$)

1. Generate a bloom filter of size m pages.
2. Insert all N keys and time how long it takes to do the operation; we will denote the elapsed time as t
3. Compute and plot the throughput: N/t

Repeat the experiment for $m = 1, 2, \ldots, 8, 9, 10, 15, 20, 25$

5.2.2 Results



These results are positive: the hierarchial bloom filter has higher throughput than the standard bloom filter.

However, I have unexplained results as my bloom filter should behave identically when $m = 1$. This will be explored for the final paper. It is possible I miss computed or made a coding error in one or both of the implementations. This requires more digging...

5.3 Measure Duration as n scales

For this experiment, we seek to validate that our implementation performs better than the standard implementation as n grows.

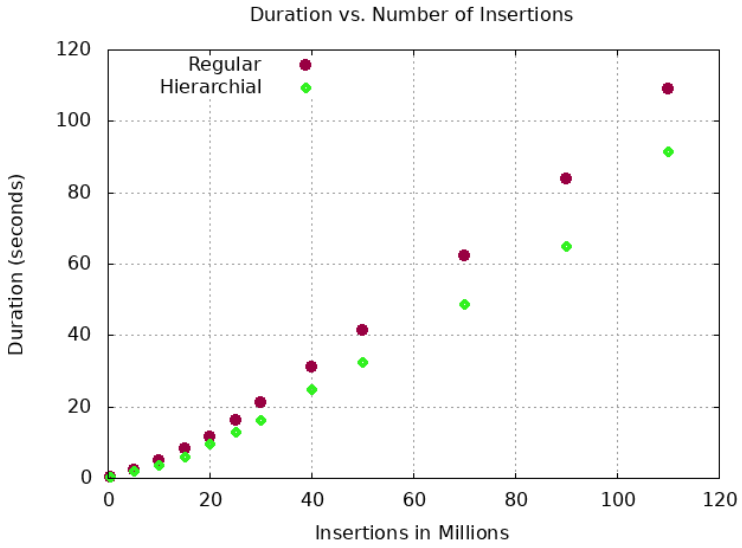
Hypothesis: *The hierarchial bloom filter will take less time to insert n keys*

5.3.1 Experimental Settings

The experiment will run as follows:

1. Generate $n = 500,000$ keys
2. Generate a bloom filter of both varianets of size $10n$. Use the optimal theoeretical configuraiton for each bloom filter (i.e, $k = 7$, $l = 1$).
3. Time how long it takes to insert all n keys into each of the bloom filters.
4. Plot the time and results.
5. Repeat for $n = 1e6, 2e6, \ldots 120e6$

5.3.2 Results



These results are positive: the hierarchial bloom filter takes noticably less time to insert n keys for every choice of n. I will do more numerical analysis for the final paper. But these results are promising!

References

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[2] Mattan Erez Evgeni Krimer. The power of 1+alpha for memory efficient bloom filters. *Electrical and Computer Engineering Department University of Texas at Austin*. URL `http://lph.ece.utexas.edu/users/krimer/pub/im11_bf.pdf`.

[3] Time Kayler. Cache efficient bloom filters for shared memory machines. *MIT Computer Science and Artificial Intelligence Laboratory*. URL `http://tfk.mit.edu/pdf/bloom.pdf`.

[4] Y.C. Tay and Min Zou. A page fault equation for modeling the effect of memory size. *Performance Evaluation*, 63(2):99–130, 2006. ISSN 0166-5316. `doi:https://doi.org/10.1016/j.peva.2005.01.007`. URL `https://www.sciencedirect.com/science/article/pii/S0166531605000180`.