Minimizing Page Faults on Bloom Filters

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 standard implementation of sufficiently large bloom filters suffers from page faults. In this paper we propose a bloom filter implementation that guarantees one page access per insert or query. This minimizes page faults, thereby drasticaly improving efficiency. We will show theoretically and empirically that our hierarchical implementation is expected to be faster than the standard implementation.
Keywords: probabilistic data structures, bloom filters, memory hierarchy, implementation, page fault analysis

1 Introduction

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. s

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 Implementations

3.1 Standard Implementation

Before we discuss the proposed solution, let us highlight the weakness of the standard implementation. The standard implementation allocates a bit vector of size m bits. Typically, this is set to be $\times 10$ the expected number of elements it will hold. The figure below represents the bit vector of length m.

Standard Bloom Filter

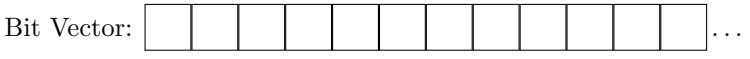


Figure 1

Per the standard implementation, to insert an element we hash the element k times and set the corresponding bit in the vector. To query, we simply read instead of set the bit. Notice, that if m is sufficiently large, it will span accross multiple pages of memory. Thus, when we read and write to the underlying bit vector, we may page fault for every bit set, slowing down our insertions or queries.

3.2 Proposed Hierarchical Implementation

Our proposal is to allocate w bit vectors of size P bits, where P is computer system's page size in bits and w is an integer such that $m = wP$. Notice, our proposed implementation uses the same amount of memory, but splits the bit vector into page size chunks.

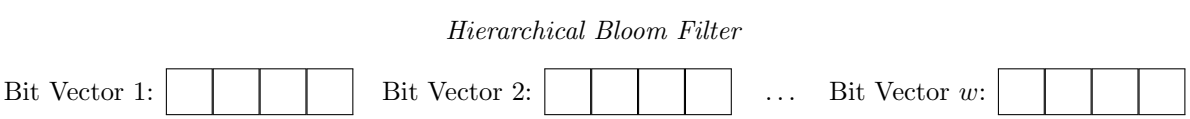


Figure 2

To insert, hash the element l times mod w and insert the element per the standard bloom filter operations to the corresponding bit vector. Each bit vector (bloom filter) has k hash functions associated with it. Here l is a pre-determined parameter of the datastructure; we will dicuss what the optimal setting is in a following section. To query, we simply query the corresponding bit vector instead of inserting. In essence, our proposal to create a bloom filter of bloom filters, hence the name.

Notice, for any given insertion or query, we have to perform l more hash operations than the standard implementation. This is not a concern if we select sufficiently cheap hash functions. More importantly, ince each bit vector is on it's own page of memory, we expect to page fault at most l times. Thus, if we minimize l without sacrificing effectiveness, we will reduce the expected number of page faults and increase the data structures efficiency.

4 Performance Theoretical Justification

5 Experiments

We will now emperically validate that the hierarchical implementation is more time efficient than the standard implementation without sacrificying accuracy. Two experiments will be conducted.

First, we will **measure elapsed time as insertions scale** of each implementation. We anticpate that the hierarchical implementation will take less time than the standard implementation for any sufficiently large value of insertions.

Second, we will **measure false positives as we scale bits allocated per element** of each implementation. We anticipate that the hierarchial and standard implementation will be approximately same as the theoerical false positive rate discussed before.

For each of these sections, we pseudorandomly generate keys to both insert and query. Discussion on exactly how these keys are generated is discussed in the appendix.

5.1 Comparing Time Efficiency

For this experiment, we seek to validate that the hierarchical implementation performs better than the standard implementation as the number of insertions grow. *Hypothesis:* The hierarchical bloom filter will take less time than the standard implementation to insert n keys for any n.

5.1.1 Experimental Settings

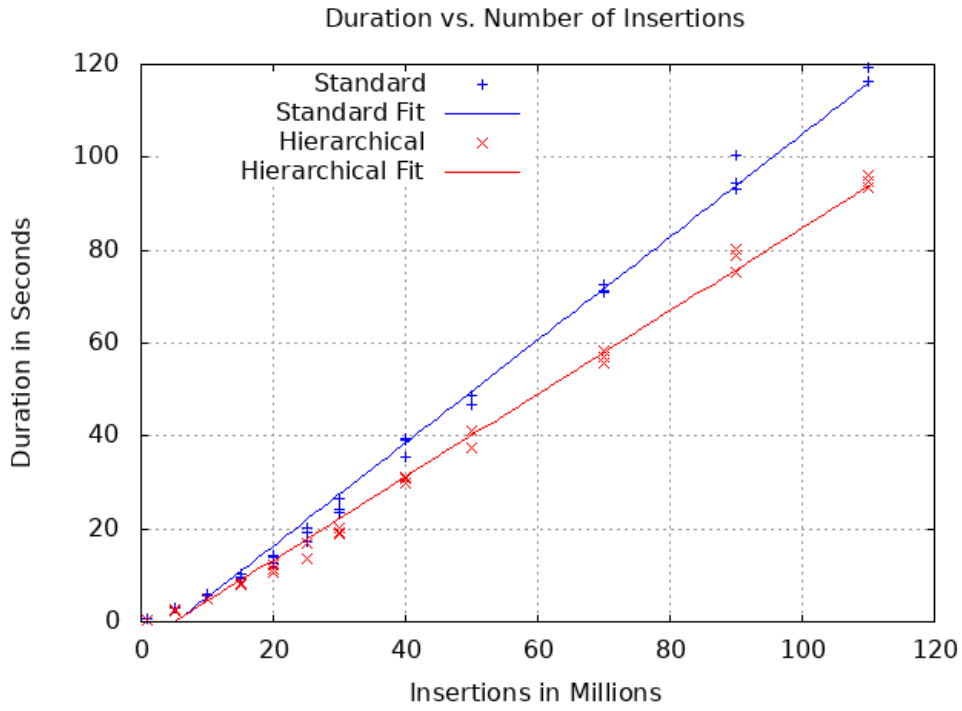
The experiment will run as follows. For each implementaiton run the following procedure:

1. Generate n random 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. Report this number.

4. Repeat for various sizes of n.

Repeat this entire process 3 times.

5.1.2 Results



The slope of the line of best fits that are plotted are as follows whre n is millions of insertions:

- The best fit line for the standard implementation is: $t = 1.07146 \cdot n - 5.83917$
- The best fit line for the hierarchical implementation is: $t = 0.789073 \cdot n - 3.64732$

Thus, our experiment shows that the hierarchical implementation performs 35% more operations per second than the standard implementation!

5.2 Comparing False Positive Rate

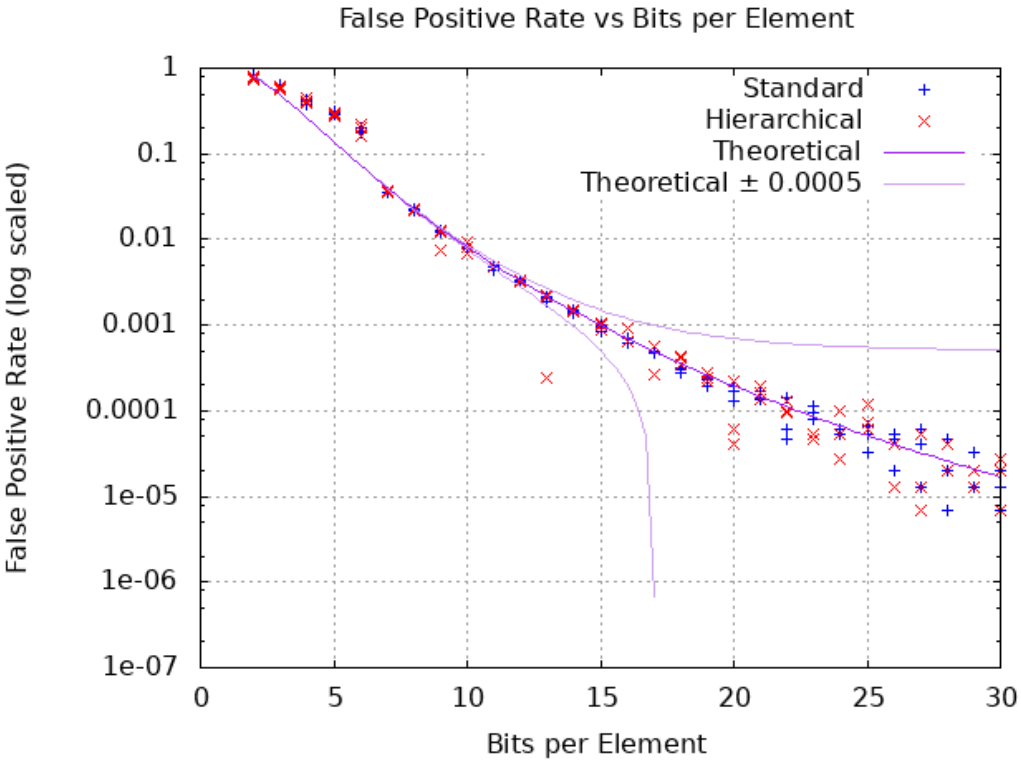
For this experiment, we seek to validate that the hierarchical implementation does not have a worse false positive rate than the standard implementation. *Hypothesis:* We expect them to have approximately the same false positive rate as the theoeretical expectation discussed earlier.

5.2.1 Experimental Settings

The experiment will run as follows.

1. Generate $150,000$ random "insertion" keys (of length 16).
2. Generate $150,000$ random "false" keys to query distinct from the insertion keys (of length 15).
3. For each implementaiton run the following procedure:
   (a) Let BPE be the bits per element (e.g $BPE = 1$ or $BPE = 10$).
   (b) Generate a bloom filter of size $150,000 \cdot BPE$.
   (c) Insert all the 'insertion" keys and query all the "false" keys and measure how many of them the bloom filter return. Report this number.
   (d) Repeat for various values of BPE.
4. Repeat this procedure again 3 times with different insertion and false keys.

5.2.2 Results



As we can see, both the standard and hierarchical implementation closely match the theoeretical expectation. Both implementations do worse than theoeretically expected if bloom filters are overpacked, but after Bits per Elements is greater than 6, both implementations are very close to theoeretical expectation ($\pm - 0.0005$) or better.

5.3 Conclusion

Our experiments have supported both of our theoretically justified hypothesises. We have demonstrated that the hierarchical implementation is more efficient than the standard implementation; it can perform 35% more operations per second! Additionally, we have verified our that our implementation is just as effective as the standard implementation.

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`.