Minimizing Page Faults of Bloom Filters

Adam Zawierucha (zawie@rice.edu)

Rice University
COMP 580: Probabilistic Data Structures & Algorithms

Abstract

Bloom filters [**?**] 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 operation. 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 without alterating the false positive rate.
Keywords: probabilistic data structures, bloom filters, memory hierarchy, implementation, page fault analysis

1 Introduction

Bloom filters are space efficient probabilistic data structures that implement set operations developed by Bloom in 1970 [**?**]. The trade-off of the afformentioned space efficiency is a probabilistic response. When an element is queried for membership, a "false" response means the element is definitely not a member, but a "true" response only means the element *may* be a member. The rate at which the data structure incorrectly reports that an element has been inserted is called the *False Positive Rate (FPR)*. This probabilistic feature allows Bloom filters to be extraordinarly space efficient, only needing a constant number of bits per stored element irrespective of the elements size.

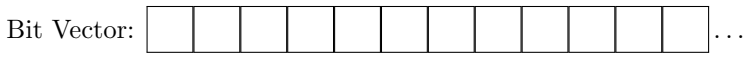
Due to Bloom filter's space efficiency they have been adopted in many domains from network security [**?**] to bioinformatics [**?**]. Any improvement in performance (without negatively alterating the false positive rate) could lead to reduce latency in network calls or speed-ups in sequenceing DNA. This motivates our proposed solution which exploits computer systems' memory hierarchy to provide better performance without increasing the false positive rate of bloom filters.

2 Implementations

2.1 Standard Implementation

Before we discuss the proposed solution, let us review 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.

Standard Bloom Filter



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 (k times), slowing down our insertions or queries.

2.2 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.

Hierarchical Bloom Filter

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, since 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.

3 Theoretical Work

In this section we will justify why the hierarchical implementation will outperform the standard implementation in theory while preserving effectiveness. First, we will **calculate expected number of page faults** for each implementation. Second, we will **find the theoeretical false positive rate**, which will guide us in our parameter selection for the implementation.

3.1 Expected Page Faults

Page faults occur when the operating system is forced to fetch data from a source lower in the memory hierarchy to be used by the process. Whenever a page fault occurs, the program must be halted to resolve the page fault, which requires relatively slow I/O operations such as checking the TLB or loading the page from disk. This leads to slower performance.

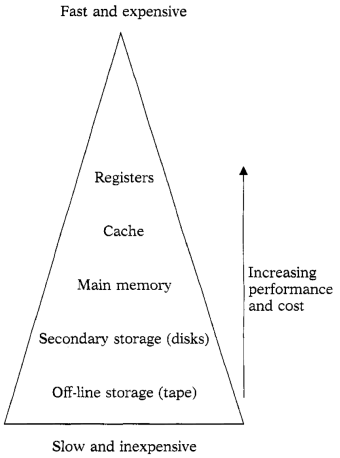


Table 1. Properties of the memory hierarchy.

Memory type	Access time	Cost/MB	Typical amount used	Typical cost
Registers	1 ns	High	1 KB	–
Cache	5–20 ns	$100	1 MB	$100
Main memory	60–80 ns	$1.10	64 MB	$70
Disk memory	10 ms	$0.05	4 GB	$200

[**?** , Murdocca et al.]

As demonstrated in the figures above from Miles J. Murdocca et al. [**?**], we see that we get radically better performance the closer we access memory to the CPU. If we design our algorithm to be more cache-efficient, then we expect massive speed-ups in practie.

Note, when discussing page faults in this section, we will only discuss the page faults caused due to accessing the underlying bit vector of the bloom filter. Natuarlly, page faults can occur while running the underlying code of the bloom filter, but this should be rare and would realistically cause at most

one page fault. We will now compute the expected number of page acceses for both the standard and hierarchical implementation.

The problem with doing an anaylsis on page faults is that it impossible to know when accessing a page will cause a page fault as this is dependant on the operating system and hardware. Generally, the more memory you are using, the higher the likelyhood a page fault will occur. In this analysis we will assume more page accesses is coorelated with a higher liklihood of page faulting. In otherwords, we can approximate the liklihood of page page fault to the expected number of different pages that will be accessed during an operation.

First, we will discuss the standard implementation. Let A be a random variable representing the number of pages accessed. Let P be the number of bits in a page and let m be the number of bits in the underlying bitvector. Suppose there are k hash functions. We now define the indictor variable A_i which is 1 if bit i is set, 0 otherwise.

$$A = A_1 + A_2 + \ldots + A_{m/P}$$

Thus, the expected number of distinct pages accessed can be found by computing the expected value that any given page is accessed and applying linearity of expectation:

$$E(A) = \sum_{i=1}^{m/P} E(A_i) = \frac{m}{P} \cdot E(A_i) \text{ for arbitrary } i$$

The probability that A_i is accesed at least once is the inverse of it being never accessed. Assuming that our hash function is uniform, we expect it to pick any particular page with probability $\frac{1}{m/P} = \frac{P}{m}$. Thus, the probablity A_i is accessed at least once is:

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

Ergo, we have a closed form equation for the expected number of page faults for a given operation:

$$\text{Expected distinct pages accessed (Standard)} = \frac{m}{P}(1 - (1 - \frac{P}{m})^k)$$

We will use this formula to compute the expected number of page faults for two reasonable cases. First, suppose you wanted a bloom filter to store $32,768$ elements with an underlying bit vector of size $\times 10$ that. Note, most computer systems have a page size of 4096 bytes, so this works out to require exactly 10 pages of memory. Additionally, suppose we pick the optimal bloom filter parameter and set $k = 7$. Under this scenario, we anticpate:

$$\text{Expected distinct pages accessed (Standard)} = 10(1 - 0.9^7) \approx 5.2$$

If we wanted to store $327,680$ elements under a similar setting, then the number of pages faults would be:

$$\text{Expected distinct pages accessed (Standard)} = 100(1 - 0.99^7) \approx 6.8$$

As we can see, even using a relatively small bloom filter, we anticpate almost every bit to be located in an entirely different page. This means we can page fault multiple times during a single operation.

We will now anaylze the hierarchical implementatin. Our implementation limits bit setting and reading to a single page per insertion or query. Therefore, we access exactly one page!

$$\text{Distinct page accessed (Hierarchical)} = 1$$

Thus, our proposed hierarchical solution page faults at most one time! Therefore, we anticpate much better performance.

3.2 False Positive Rate

In this section we will demonstrate that both implementations will have the same false positive rate in theory.

3.2.1 Standard False Positive Rate

As we have covered in class, the false positive rate of a standard bloom filter is as follows.

$$(1 - (1 - \frac{1}{m})^{nk})^k \approx (1 - e)^{kn/m} \tag{1}$$

Setting $k = 7$ will minimize the false positive rate, as discussed in class.

3.2.2 Hierarchical False Positive Rate

Suppose we have a allocated m bits in total chunked into bit vectors of size P and that we have to inserted n elements into our bloom filter.

We now want to compute the false positive rate of the hierarchical implementation. This can be computed by supposing the bloom filter has been filled with n elements and computing the probability that a querying a false key would result in a true response.

Assuming our hash function is uniform, any particular sub-bloom filter is anticpated to have $\frac{nl}{m/P} = \frac{nlP}{m}$ elements in it. There are m/P bloom filters and we choose l of them to check. As before k is the number of hash functions a particular bloom filter uses. Thus, the chance any one bloom filter is set is:

$$(1 - (1 - \frac{1}{P})^{lknP/m})^k$$

For our hierarchical implementation to return true, all l bloom filters selected must return a positive result. Thus, the false positive rate is:

$$((1 - \frac{1}{P})^{lknP/m})^{kl}$$

We can use a well known identity for e^{-1} to approximate this false positive rate:

$$\approx (1 - e^{-lkn/m})^{lk}$$

Interestingly, this removes the dependence on the page size. However, it is important to note that the Euler identity is $e^{-1} = \lim_{x\to\infty}(1 - \frac{1}{P})^P$, so this identity fails as an approximation the smaller P becomes. Therefore, we should pick P to be the largest value where we expect speed ups, which would be the size of a physical page. More importantly, this formula becomes isomorphic to the false positive rate for the standard implementation. If we let $k' := lk$ we see that our false positive rate for the hierarchical implementation is:

$$\approx (1 - e^{-k'/m})^{k'}$$

Therefore, as discussed for the standard implementation, the optimal selection for k' is 7. Therefore, either $l = 1$ and $k = 7$ or $l = 7$ and $k = 1$. Since our goal is reduce the number of pages accessed, the former is a more sensible choice.

Therefore, the best parameter selection for our hierarchical implementation would be to select 1 bloom filter the size of a bloom filter and set 7 bits in each one.

$$l = 1 \ ; \ k = 7$$

3.3 Conclusion

Our theoretical work has justified the claim that we expect this implementation to be more performant while having the same positive rate as the standard implementaiton. Moreover, we have found the optimal parameters to use for our implementation.

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

4.1 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.**

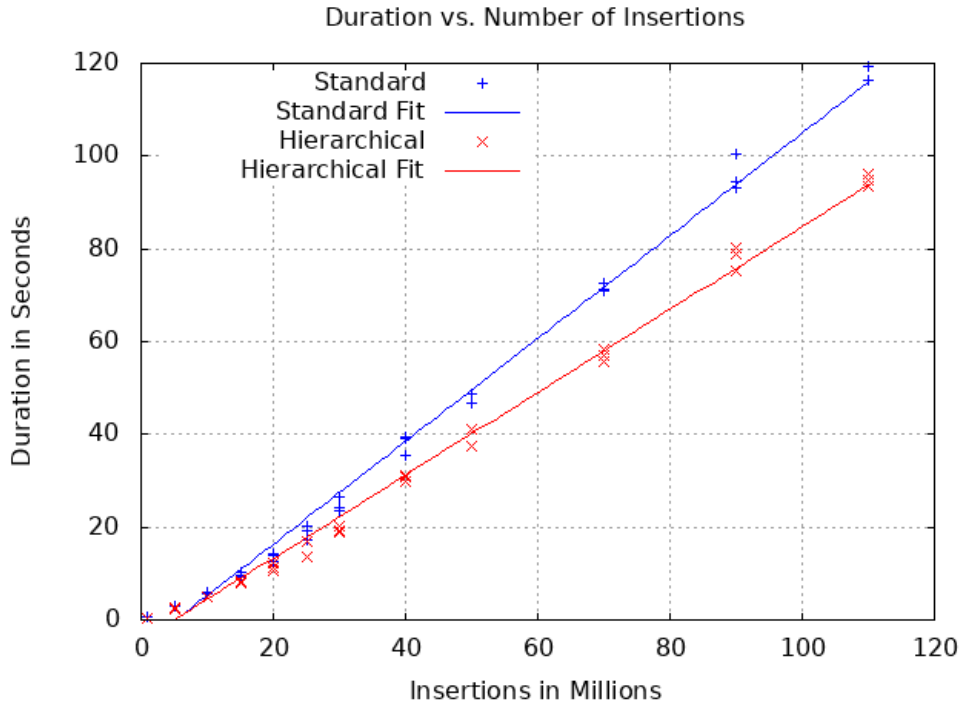
4.1.1 Experimental Settings

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.

4.1.2 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: $t = (1.1078 \pm 0.01446) \cdot n - (5.78611 \pm 0.7407)$
- The best fit line for the hierarchical implementation is: $t = (0.893221 \pm 0.01166) \cdot n - (4.52759 \pm 0.597)$

We will disregard the constant as we care about how these data structures perform as input volume scales. We can compute the efficiency difference by dividing the slopes:

$$\text{Hierarchical Implementation Slope/Standard Implementation Slope} = \text{Efficiency Difference}$$

$$(0.893221 \pm 0.01166 \text{ seconds/operation})/(1.1078 \pm 0.01446 \text{ seconds/operation}) = 0.806301679 \approx 80\%$$

In other words, our implementation takes 80% less time per operation to complete. Thus, for any time frame, if the standard implementation performs one operation, the hierarchical implementation is expected to perform $1/80\% = 1.25$ operations. Thus, our experiment shows that the hierarchical implementation performs 25% more operations per second than the standard implementation!

4.2 Comparing False Positive Rate

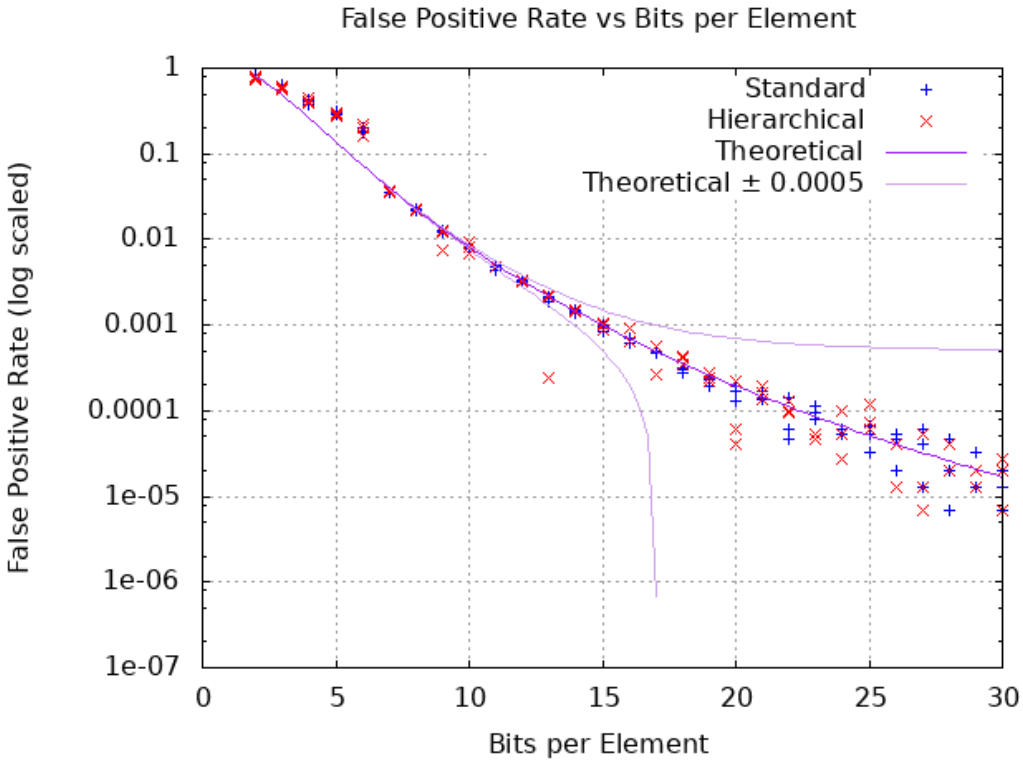
For this experiment, we seek to validate that the hierarchical implementation does not have a worse false positive rate than the standard implementation. **Hypothesis: We expect them to have approximately the same false positive rate as the theoeretical expectation discussed earlier.**

4.2.1 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 as being a member. Report this number.
   (d) Repeat for various values of BPE.
4. Repeat this procedure again 3 times with different insertion and false keys.

4.2.2 Results



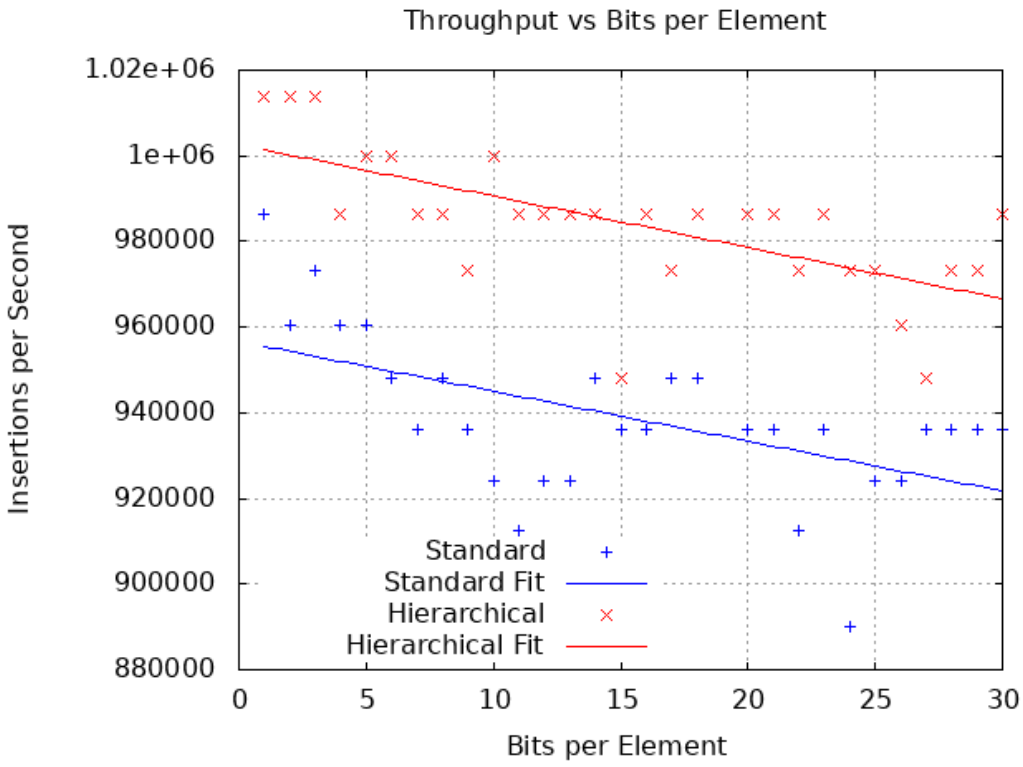
As we can see, both the standard and hierarchical implementation closely match the theoeretical expectation. Both implementations do worse than theoeretically expected if bloom filters are overpacked, but after Bits per Elements is greater than 6, both implementations are very close to theoeretical expectation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

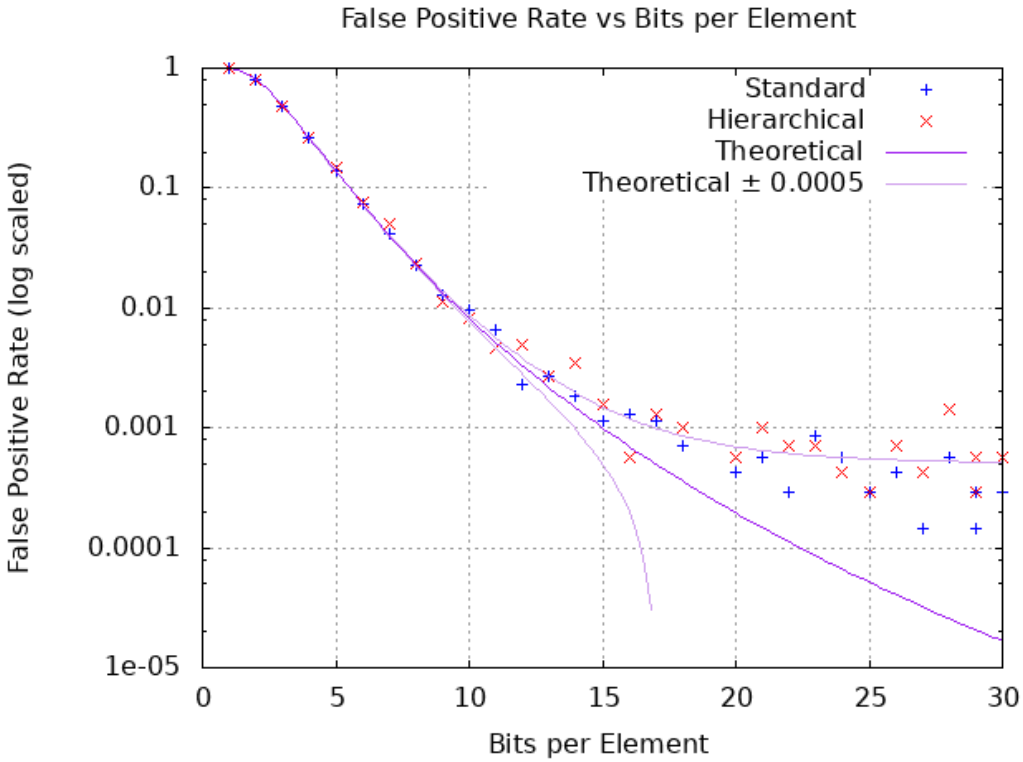
4.3 Conclusion

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 25% more operations per second! Additionally, we have verified our that our implementation is just as effective as the standard implementation.

5 Application

We will now compare the standard and hierarchical implementation, against a real data set.





6 Literature Survey

Note: I decided to do a literature survey of different data structures solving this problem *after* I explored this idea. I wanted to have the joy of discovering whether or not my idea would work on my own! Felix Putze et al. developed bloom filters with better cache efficiency and requiring less hash bits than the standard bloom filter [**?**]. One of their implementations is similar to mine. They discretize a bloom filter into *cache size* (64 bytes) chunks instead of *page size* chunks (4096 bytes). This leads to even *faster* improvements at the cost of a higher false positive rate. Remember, the smaller you make the sub-bloom filter, the worse the Euler-identity based approximation becomes. They also recompute random bit patterns: instead of setting k bits in the bloom filter, they hash once to create a mask that sets multiplem bits [**?**]. This saves them time while performing hash functions. The combination of these techniques gave them much better performance over the standard implementation with slight false positive rate increases. I would be interested in further exploring if some pattern masking can be applied to our bloom filter. This may require another level in our hirarchy to give us small filters that we can efficiently mask over.

Rafael P. Laufer et al. developed a "tamper proof" Generalized Bloom Filter [**?**] to counteract an inherent vulnerability in bloom filters. The authors call the exploitation of this vulnerability an "all one" attack, where a malicious agent floods the bloom filter with requests to set all the bits to one, artificially increasing the false positive rate. This may slow down the program that the bloom filter is employed in and marks elements as members of the set at a higher rate than it should. While this paper does not seek to solve a similar problem to our implementaiton, it is important to note that the hierarchy we imposed on our bloom filter may make our implementation more susecptible to this kind of attack. If a malicious agent wanted to cause a high false positive rate for a certain kind of element, they need not find a system to set all the bits. Instead, they would only have to exploit one hash function: the hash function that selects bloom filters. This may pose a security risk. Thus, for production uses, it may be wise to use a cryptographically secure hashing algorithm for this "choke point" hash function. However, we could also employ technieques discussed by Laufer, such as having multiple hash functions that set and reset bits accross the bit vector [**?**]. In any case, further research should be done to prevent our implementation from being exploited.

7 Conclusion

In this paper, we have theoretically and emperically justified a variant implementation of bloom filters that lead to much higher performance without increasing the false positive rate. We analyzed the expected number of unique pages accessed per query, which should coorelate with expected number of page faults, and found that our implementation is expected to page fault less. Moreover, we found a closed form expression of the false positive rate of our implementaiton and found that it was equivalent to the standard implementation's false positive rate. Additionally, we performed two experiments with randomly generated keys to emperically back these claims. Lastly, we compared the standard and hierarchical implementation on a real data set. Our findings were positive: our implementation is more efficient than the standard implementation while maintaing the same false positive rate.

8 Appendix

8.1 Code

8.2 Raw Data

Here I will attach the raw data generated to create the efficieny and false positive graphs used earliy

8.2.1 Duration vs. Number of Insertions

Number of Insertions (Millions)	Standard (Seconds)	Hierarchical (Seconds)
1.000000	0.560000	0.460000
5.000000	2.880000	2.590000
10.000000	5.700000	4.820000
15.000000	9.240000	8.210000
20.000000	12.610000	11.430000
25.000000	17.180000	13.740000
30.000000	23.660000	19.330000
40.000000	39.500000	30.800000
50.000000	46.850000	37.410000
70.000000	71.260000	55.800000
90.000000	93.240000	75.390000
110.000000	116.370000	93.550000
1.000000	0.550000	0.460000
5.000000	2.830000	2.370000
10.000000	6.070000	5.020000
15.000000	10.400000	7.890000
20.000000	13.800000	12.310000
25.000000	20.190000	16.750000
30.000000	26.430000	18.900000
40.000000	39.220000	31.150000
50.000000	48.870000	41.220000
70.000000	72.440000	58.360000
90.000000	100.580000	78.930000
110.000000	116.370000	96.090000
1.000000	0.570000	0.450000
5.000000	2.870000	2.390000
10.000000	5.850000	4.990000
15.000000	9.620000	7.810000
20.000000	14.240000	10.770000
25.000000	19.080000	17.060000
30.000000	24.240000	20.110000
40.000000	35.450000	29.870000
50.000000	46.760000	37.400000
70.000000	71.070000	56.980000
90.000000	94.480000	80.160000
110.000000	119.380000	94.760000

8.2.2 False Positive Rate vs. Bits Per Element

Bits per Element	Standard (Seconds)	Hierarchical (Seconds)
2	0.825093	0.767273
3	0.644347	0.605580
4	0.368693	0.447120
5	0.305813	0.272167
6	0.203047	0.204920
7	0.035727	0.036307
8	0.021687	0.022007
9	0.012247	0.012513
10	0.007927	0.009440
11	0.004360	0.004827
12	0.003413	0.003240
13	0.002133	0.002120
14	0.001440	0.001420
15	0.001027	0.000880
16	0.000613	0.000640
17	0.000473	0.000260
18	0.000273	0.000433
19	0.000233	0.000273
20	0.000127	0.000060
21	0.000133	0.000160
22	0.000047	0.000100
23	0.000093	0.000047
24	0.000053	0.000100
25	0.000067	0.000120
26	0.000020	0.000040
27	0.000013	0.000007
28	0.000047	0.000020
29	0.000013	0.000020
30	0.000020	0.000020
2	0.824393	0.744333
3	0.640540	0.592980
4	0.425847	0.397867
5	0.315887	0.299433
6	0.172600	0.160847
7	0.035480	0.035833
8	0.021773	0.022273
9	0.012673	0.007560
10	0.008173	0.008227
11	0.005080	0.004880
12	0.003160	0.003173
13	0.001840	0.002227
14	0.001507	0.001427
15	0.000833	0.000987
16	0.000620	0.000913
17	0.000500	0.000267
18	0.000300	0.000413
19	0.000240	0.000220
20	0.000167	0.000040
21	0.000173	0.000133
22	0.000140	0.000093
23	0.000113	0.000053
24	0.000060	0.000053
25	0.000033	0.000060
26	0.000047	0.000000
27	0.000040	0.000053
28	0.000007	0.000040
29	0.000013	0.000020
30	0.000007	0.000027
2	0.832767	0.804993
3	0.645093	0.572180
4	0.409427	0.401713
5	0.286507	0.286600
6	0.180753	0.217647
7	0.035460	0.037447
8	0.022280	0.021473
9	0.012967	0.012333
10	0.008047	0.006773
11	0.004813	0.004693
12	0.003200	0.003413
13	0.002033	0.000247
14	0.001347	0.001487
15	0.000960	0.001033
16	0.000720	0.000647
17	0.000487	0.000553
18	0.000320	0.000347
19	0.000193	0.000247
20	0.000193	0.000220
21	0.000140	0.000193
22	0.000060	0.000127
23	0.000080	0.000053
24	0.000053	0.000027
25	0.000053	0.000073
26	0.000053	0.000013
27	0.000060	0.000013
28	0.000020	0.000020
29	0.000033	0.000013
30	0.000013	0.000007

8.2.3 Throughput vs. Bits Per Element (AOL)

Bits per Element	Standard (Insertions/Second)	Hierarchical (Insertions/Second)	Standard False Positive Rate	Hierarchical False Positive Rate
1	986486.486486	1013888.888889	0.994547	0.994404
2	960526.315789	1013888.888889	0.803272	0.799828
3	973333.333333	1013888.888889	0.479409	0.484001
4	960526.315789	986486.486486	0.263022	0.259291
5	960526.315789	1000000.000000	0.139618	0.149806
6	948051.948052	1000000.000000	0.074186	0.077199
7	935897.435897	986486.486486	0.041326	0.050940
8	948051.948052	986486.486486	0.022672	0.023533
9	935897.435897	973333.333333	0.012627	0.011192
10	924050.632911	1000000.000000	0.009614	0.008323
11	912500.000000	986486.486486	0.006601	0.004592
12	924050.632911	986486.486486	0.002296	0.004879
13	924050.632911	986486.486486	0.002726	0.002726
14	948051.948052	986486.486486	0.001865	0.003444
15	935897.435897	948051.948052	0.001148	0.001578
16	935897.435897	986486.486486	0.001291	0.000574
17	948051.948052	973333.333333	0.001148	0.001291
18	948051.948052	986486.486486	0.000717	0.001004
29	935897.435897	973333.333333	0.000143	0.000287
20	935897.435897	986486.486486	0.000430	0.000574
21	935897.435897	986486.486486	0.000574	0.001004
22	912500.000000	973333.333333	0.000287	0.000717
23	935897.435897	986486.486486	0.000861	0.000717
24	890243.902439	973333.333333	0.000574	0.000430
25	924050.632911	973333.333333	0.000287	0.000287
26	924050.632911	960526.315789	0.000430	0.000717
27	935897.435897	948051.948052	0.000143	0.000430
28	935897.435897	973333.333333	0.000574	0.001435
29	935897.435897	973333.333333	0.000287	0.000574
30	935897.435897	986486.486486	0.000287	0.000574