An Analysis of Cuckoo Filter Performance in Different Performance Context

Minh Le and Frank We

I. ABSTRACT AND INTRODUCTION

The Cuckoo filter, a new data structure proposed by Fan, et. al (2014), is centered around the utilization of Cuckoo hashing for approximate set membership tests with high accuracy. Similar to its predecessor, the famous Bloom filter, this data structure features four main benefits: a zero false negative rate, constant lookup time, efficient space utilization and low false positive rate. While Bloom filter has theoretically faster insertion, Cuckoo filter has a lower false positive rate. Furthermore, the latter also allows for easy dynamic deletion whereas simple deletion in the former would introduce a possibility of false negatives. Given its relatively new introduction, not much literature is available on the performance of Cuckoo filter.

In this project, we aim to address this problem and implement an efficient Cuckoo filter, and then provide a look into the performance capability of the Cuckoo filter, as well as its possible application in a parallel context. We have built a new parallel implementation of a Cuckoo filter and a Bloom filter in Python. These multithreaded implementations take in the number of threads, and produce corresponding performance graph. The Bloom filter is used as a baseline for performance comparison, and our parallel Cuckoo filter has exhibit an increase in performance as number thread increase but also increase in space usage by 20%. Both filters utilize murmurhash3 as their internal hash function.

II. BACKGROUND AND LITERATURE REVIEW

The Cuckoo filter takes advantage of the novel Cuckoo hashing method. Chen (n.d.) provides an overview of Cuckoo hashing, which is a form of hashing that can resolve collision in constant time worst case lookup and deletion and amortized constant time insertion. Cuckoo hashing utilizes two hash tables T_1 and T_2 , each with its own corresponding hash function (h_1 and h_2 respectively) and r buckets. Look up is a simple check of the existence of $h_1(x)$ in T_1 or $h_2(x)$ in T_2. Deletion is also straight forward. Look up the element and delete the element if the element exists. This is already an improvement compared to the Bloom filter because in the Bloom filter, one cannot simply delete element in the bit array. On the other hand, insertion and collision handling are a bit more complicated. Hashing a new item x is simply inserting x to either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. It is worthy to note that x can only be in one of the two tables but not both. However, there already may be an existing value y in the hash. If this is the case, the value y will be evicted from that bucket and needs to be rehashed into the other table. This

procedure of replacing and rehashing evicted values continue until a key is inserted in an empty slot. If this does not resolve within a set number of time steps, the whole table size is increased and all keys are rehashed. Chen (n.d.) mentioned the choice of hash function significantly impacts Cuckoo hashing performance. Compared to other hashing schemes like two-way hashing, chaining, linear probing, Cuckoo hashing performed slower in insertion, fast in lookup, and fastest in deletion.

As mentioned previously, Cuckoo filter supports adding and removing items dynamically whereas removing in Bloom filter is not a viable option. Fan et. al (2014), who introduced Cuckoo filter, has showed that Cuckoo Filter has lower space overhead than space-optimized Bloom filter in applications with a moderately low false positive rate (less than 3%). Interestingly Cuckoo filter only stores fingerprints, which are bit string derived from the item using a hash function. Consequently, the false positive rate of Cuckoo filter decrease as size of fingerprint increases logarithmically. However, due to the usage of fingerprints, rehashing the element in the insertion of Cuckoo hashing is not viable in the Cuckoo filter. Thus, Fan et. al (2014) employ partial key Cuckoo hashing to overcome this limitation. Partial key Cuckoo hashing is a variant scheme of standard Cuckoo hashing, using fingerprints stored in the hash to get to next location: $h_1(x) = hash(x)$ whereas $h_2(x) = h_1(x)$ *XOR hash(finge*rprint of x). Fan et. al's implementation of the Cuckoo filter has achieved a false positive rate of 0.19% with 2^{25} buckets per hash tables and only 12.6 bits per item, whereas their implementation of Bloom filter has achieved the same positive rate, but required 13 bits per item.

While there is not a lot of literature on parallel implementation of Cuckoo filter, Alcantara. et. al (2009) discusses the performance of parallel Cuckoo Hashing which gives an insight into the potential parallel usage of the Cuckoo filter. Their results display a scalability issue. This method is not good with large hash tables on GPU, but for small and fast on-chip memory computation, it proves to be useful. However, construction times are similar and look up times are better compared to "the approach of representing sparse data in an array using a state-of-the-art [sorting method]" (Dan. et. al 2009). This motivates us further to explore the Cuckoo filter in a parallel setting.

III. HYPOTHESIS

We hypothesize that we can experience significant improvement on performance and memory usage using Cuckoo filter instead of standard Bloom filter in both single threading and multithreading context.

IV. EXPERIMENT AND METHODOLOGY

The experiment is conducted using Python 3. For our threading implementation, we used Python innate threading library *thread*. We used both the *murmurhash* and *mmh3* as our hash library. For consistency purposes across different tests, we always sampled from the same dataset deterministically, which include non-member set and member set as well as using a seed for *murmurhash* and *mmh3*. We also used the *bitarray* library for better memory management and efficiency. All library need to be installed in order to run the script file without error. Please consult the attached README for further installation instruction.

The implementation procedure is straightforward. As the base for development, the Bloom filter was used and then parallelized using the *th*read library. For the Cuckoo filter, we based our implementation on FastForward Lab's sequential Cuckoo filter implementation (Retrieved from https://github.com/fastforwardlabs/cuckoofilter). This implementation of the Cuckoo filter was used because it showed much better performance and correctness compared to our original attempt at implementing a Cuckoo filter. Then we also parallelized the Cuckoo filter.

As for testing, we perform several performance tests and memory usage tests of parallel version of the Cuckoo filter and the Bloom filter. We tested both filters on different numbers of threads (1,2,4, and 8) to see the difference. We also used two computers with different processing power to see the difference. Specifically, the first set of tests was run on a MacBook with a 2.7 GHz Intel Core i5 processor and the second set of tests was run on a MacBook Pro with a 2.9 GHz Intel Core i7 processor.

Due to the inherent arbitrary assignment of threads and computation power assigned to Python by macOS and the *thread* library, the test results may turn out slightly different on a very static environment. However, the overall trend should still be displayed. Moreover, if other background processes or applications are running alongside Python, the performance may turn out vastly different among each result. Thus, it is advised that in order to replicate the test results, no other processes should be running.

We also tested for false positive and false negative rates to compare possible error rate each filter provides depending on how much memory space is allocated to each filter. These tests are already included within the provided code, and all tests should show as plots if run according to instructions in the attached README document.

V. EXPERIMENTAL RESULTS

1. Performance and Memory Difference between Non Parallel Implementations

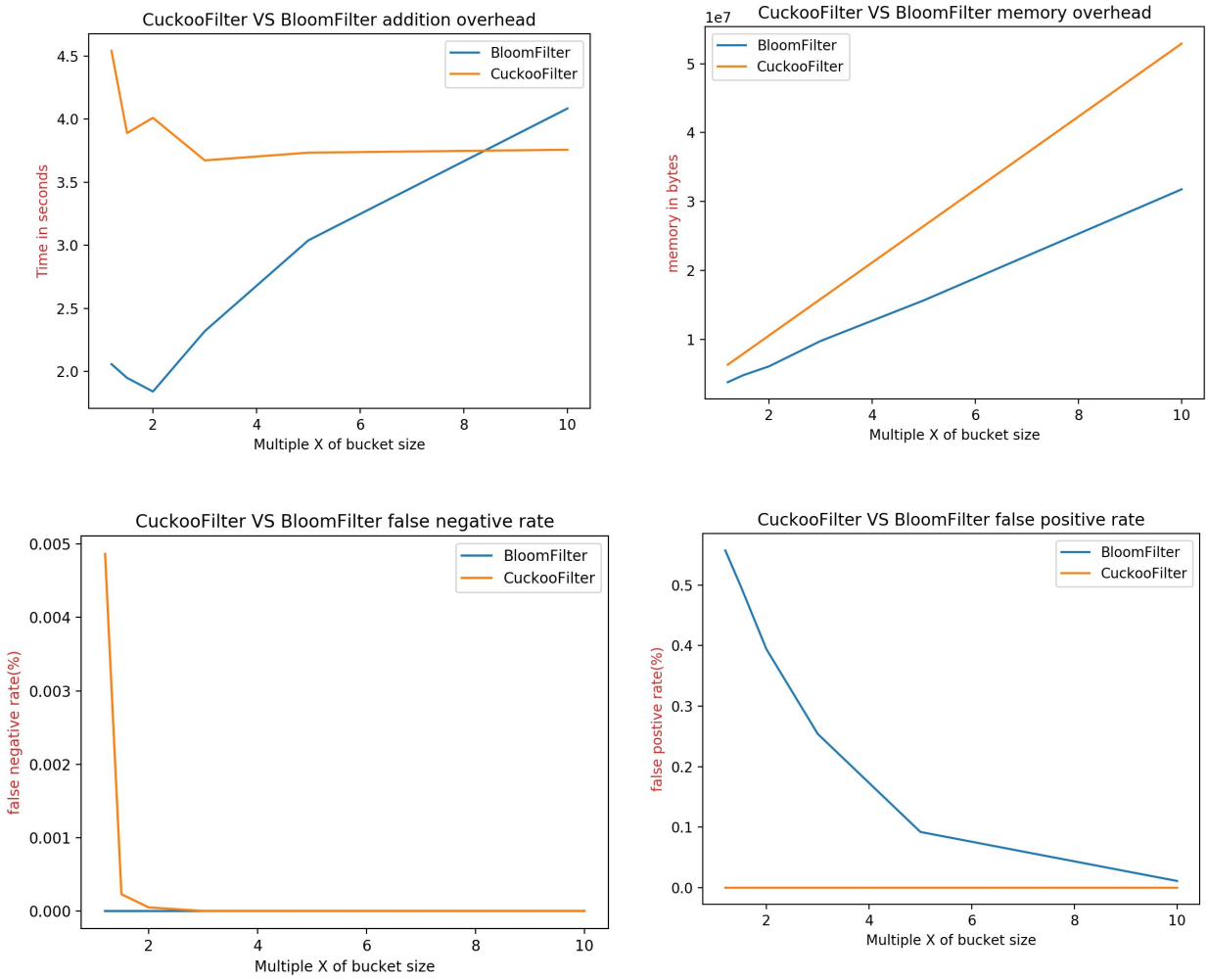


Figure 1. Plots of single threaded Cuckoo filter versus single threaded Bloom filter with regards to addition overhead, memory usage, false positive and negative rate

These test results are obtained by setting the number of thread the parallel implementations used to 1, which is equivalent to using a sequential Cuckoo filter and a sequential Bloom filter. There are a few interesting analysis from these plots. Firstly, as the bucket size increases Cuckoo filter starts to perform better than Bloom filter. Secondly, memory overhead is slightly higher for Cuckoo filter because it needs to keep track of fingerprints as well as having two internal hash tables. The false negative rate is zero for Bloom filter whereas

Cuckoo filter has a very small chance of having a false negative rate. However, this is expected due to the way fingerprints are formed.

As can be observed from this plot, Cuckoo filter displayed better false positive rate, which is zero in this experiment, than Bloom filter. These results conform to finding in previous literature and proves the correctness of our implementation.

2. Comparison of the Cuckoo Filter insertion performance and the Bloom Filter insertion performance with multiple threads using the 2.7 GHz Intel Core i5 processor (4 threads).

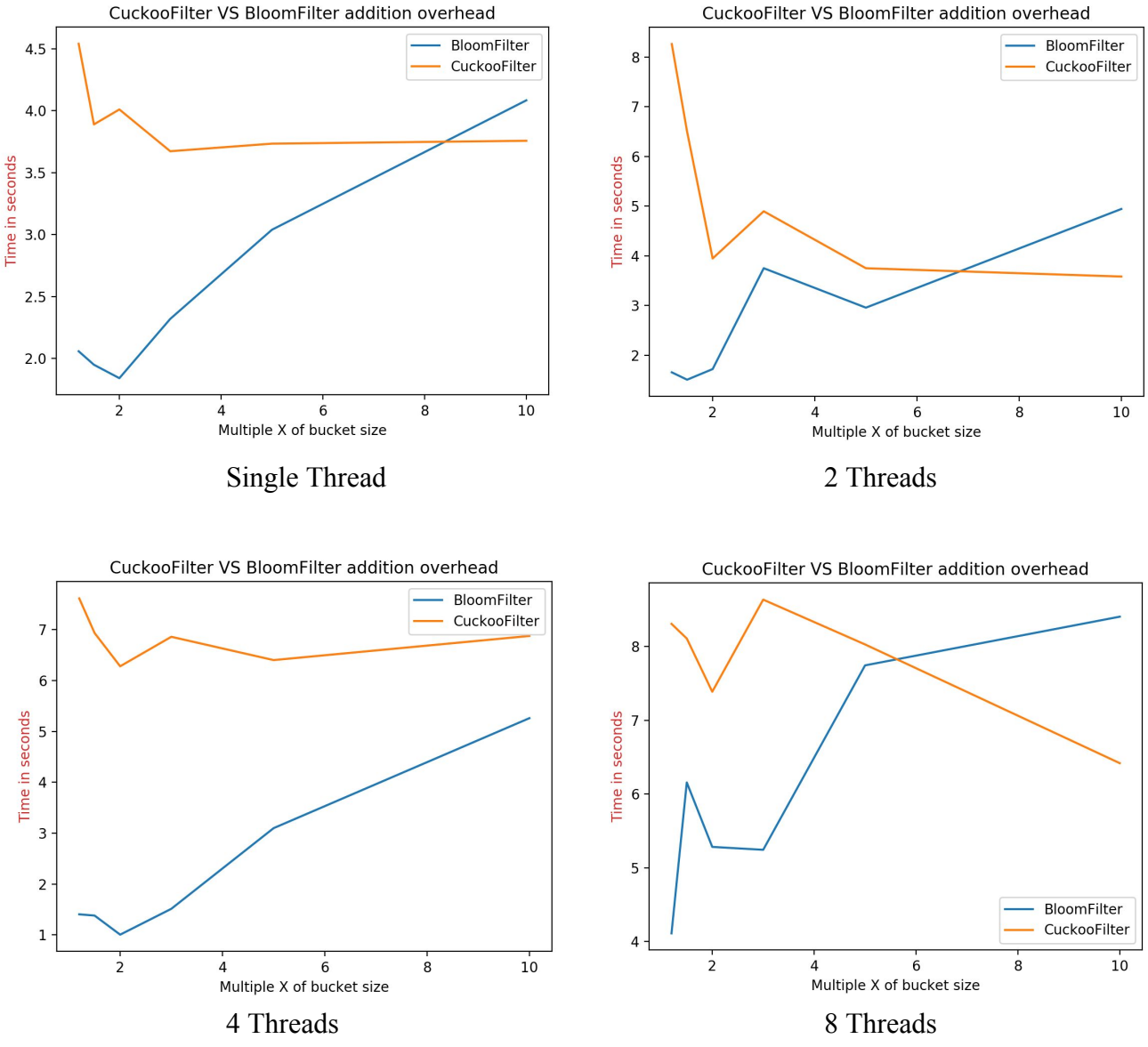


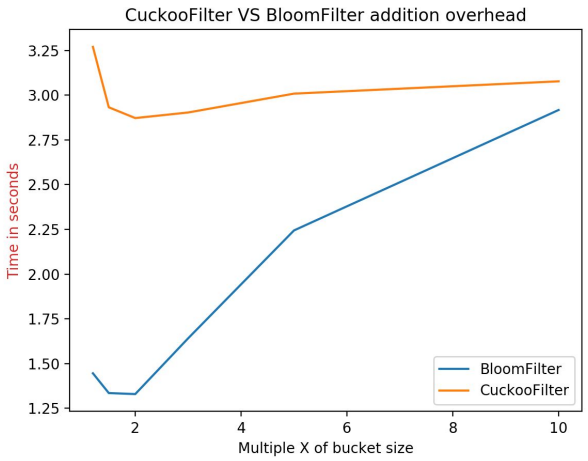
Figure 2. Plots of multithreaded Cuckoo filter versus multithreaded Bloom filter with regards to insertion performance using a 2.7 GHz Intel Core i5 processor.

Since memory, false positive, and false negative rate are all very similar in the single threading case and the multithreading case, we chose to focus more on the cost of insertion in the parallel context. In other words, the overhead of addition appears to be a good criterion to compare the difference between the performance of the Cuckoo filter and that of the Bloom filter. We initially tested on the 2.7 GHz Core i5 processor, which has 4 threads internally. Plots show that the Cuckoo filter starts to increase its performance compared to the Bloom filter as the number of threads increase.

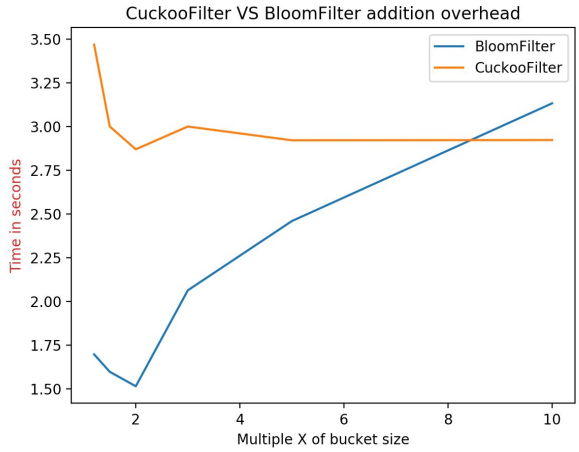
It is worth to note that the test environment is very fragile and rather nondeterministic because it is dependent on Python inner threading implementation and how much CPU is willing to allocate to the python threading. Nevertheless, we can see that the performance of the Cuckoo filter surpasses that of the Bloom filter much quicker overall as we increase threads count. To provide further proof of this observation, we decided to test with a different processor.

Furthermore, one reason for the plot of 4-threading being so different may be due to the fact that it is trying to use the maximum number of threads the processor has. Therefore it cannot actually attain all 4 threads and result in an anomaly in performance. In the case of 8-threading, more than half of the threads will not be allocated at all time because there simply are not enough threads to handle it. That is why the overhead is larger compared to other cases.

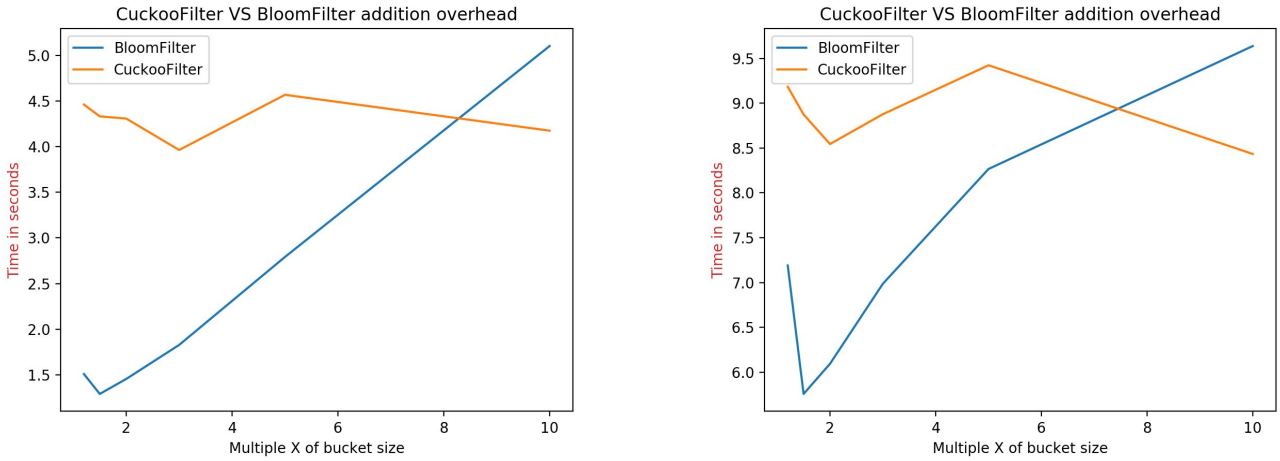
3. Comparison of the Cuckoo Filter insertion performance and the Bloom Filter insertion performance with multiple threads using the 2.9 GHz Intel Core i7 (8 threads)



Single Thread



2 Threads



4 Threads 8 Threads

Figure 3. Plots of multithreaded Cuckoo filter versus multithreaded Bloom filter with regards to insertion performance using a 2.9 GHz Intel Core i7 processor.

In the next set of tests, we obtained results on a 2.9 GHz Intel Core i7 processor, which has 8 threads internally. These new plots further depict that the Cuckoo filter indeed outperforms the Bloom filter with increasing threads count, as the two performance lines converge faster the more threads they use.

VI. CONCLUSION

We have found that with multiple threading we can achieve significant improvement in performance of the Cuckoo filter compared to that of the Bloom filter. Thus, we have managed to prove that the Cuckoo filter performs better than the Bloom filter in both single threading and multithreading context. However, the trade-off of using a Cuckoo filter instead of a Bloom filter is the exchange of false positive rate and false negative rate, the latter of which is non-existent in a Bloom filter. Yet the steeper decrease of error rate for the Cuckoo filter and the flexibility for dynamic addition and insertion as well as better performance in a parallel context have led us to believe that the Cuckoo filter is a better candidate in the ever growing Big Data world. We conclude that we can achieve a new form of maintaining hash much more efficient than regular Bloom filter and that parallel Cuckoo filter proves to be a viable alternative to Bloom filter for membership-test application. We believe that with further research, innovative applications of parallel Cuckoo filters can be found and applied in a meaningful way to large scale data set.

VII. REFERENCE

Fan, B., Andersen, D. G., Kaminsky, M., & Mitzenmacher, M. D. (2014). Cuckoo Filter: Practically Better Than Bloom. Retrieved January 24, 2019, from https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

Chen, Charles. "An Overview of Cuckoo Hashing." (n.d.). Retrieved January 24, 2019, from https://cs.stanford.edu/~rishig/courses/ref/l13a.pdf.

Alcantara, Dan A., Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. "Real-time parallel hashing on the GPU." ACM Transactions on Graphics (TOG) 28, no. 5 (2009): 154.