Birdwatching: False Negatives In Cuckoo Filters

Novak Boskov
Boston University
Boston, MA, USA
boskov@bu.edu

Ari Trachtenberg
Boston University
Boston, MA, USA
trachten@bu.edu

David Starobinski
Boston University
Boston, MA, USA
staro@bu.edu

ABSTRACT

Cuckoo filters are a probabilistic data structure for approximate set membership queries. Their lookup queries are designed to return either "probably in the set" (with probability of error ϵ) or "definitely not in the set". We show that the latter does not necessarily hold in practice, meaning that these filters may suffer from both false positives and false negatives. Specifically, we analyze state-of-the-art cuckoo filter implementations, and identify a source of false negatives arising from an interplay between partial-key hashing and cuckoo evictions in filters that are close to full. We further show that for practical implementations of cuckoo filters, there is a trade-off between space efficiency and incurring a certain amount of false negatives. Finally, we compare state-of-the-art cuckoo filter implementations with their Bloom filter counterparts. We show that for a false positive rate below 3%, Bloom filters achieve better space efficiency than cuckoo filters for most of the filter sizes.

CCS CONCEPTS

• **Theory of computation → Bloom filters and hashing**;

KEYWORDS

Cuckoo filters, Bloom filters

ACM Reference Format:
Novak Boskov, Ari Trachtenberg, and David Starobinski. 2020. Birdwatching: False Negatives In Cuckoo Filters. In *Student Workshop (CoNEXT'20), December 1, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3426746.3434063

1 INTRODUCTION

Cuckoo filters are an approximate set membership data structure that support *Insert*, *Lookup*, and *Delete* operations. They were introduced by Fan et al. [4] in 2014, and have since been adopted in a wide range of network applications in which Bloom filters were utilized beforehand. To maximize their space efficiency and operations throughput (*i.e., number of operations per second*), cuckoo filters make their *Lookup* approximate, admitting a fraction ϵ of false positives. By design, cuckoo filters are supposed to not incur any false negatives for *Lookup*.

Our contributions are summarized as follows:

CoNEXT'20, December 1, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-8183-3/20/12…$15.00
https://doi.org/10.1145/3426746.3434063

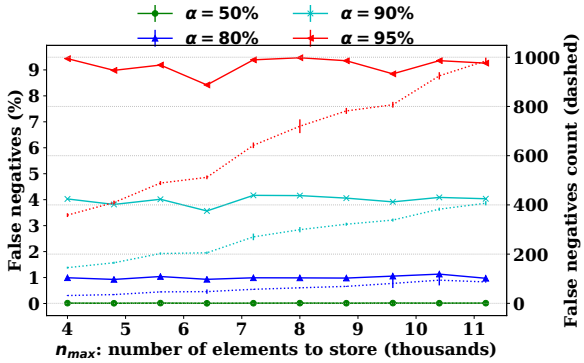

Figure 1: Empirical false negatives rate for $b = 4, f = 8$ cuckoo filters, with m not a power of two.

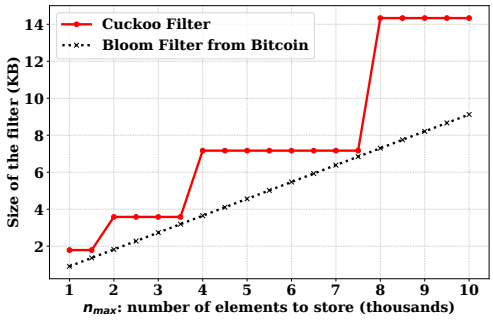

Figure 2: Space utilization of space optimized cuckoo [1] and Bloom filters [2].

- We conduct experiments to measure the space efficiency of cuckoo filters [1], showing that they may use up to twice the theoretically expected amount of memory.
- We compare the space efficiency of cuckoo filters [1] with Bloom filters implementation in Bitcoin Core [2] and identify conditions under which Bloom filters are smaller than cuckoo filters (See Fig. 2).
- Finally, we identify the potential source of false negatives in some cuckoo filters and evaluate their effect in practical filter configurations (See Fig. 1).

2 STRUCTURE OF CUCKOO FILTERS

A cuckoo filter is a two-dimensional bit array divided logically into slots, buckets and fingerprints. The number of slots per bucket (b), number of buckets (m), and bits per slot (f) are fixed and uniform throughout the filter. We write n_{max} to denote the maximum capacity of the filter, n to denote the number of currently inserted elements, and $\alpha = n/(b * m)$ to denote the current load factor. Cuckoo

filters also make use of two independent hash functions: the *fingerprint hash* (x_f) and *bucket hash* (H). These two hash functions participate in *partial-key cuckoo hashing* — the procedure of identifying the two buckets into which a fingerprint ϕ_x of an item x may be stored. The fingerprint hash determines the fingerprint of an item, while the bucket hash determines the two *candidate buckets* (denoted i_1 and i_2) into which the fingerprint may be stored. The fingerprint of each successfully inserted item is stored in only one of the two candidate buckets, depending on which bucket has sufficient space. If no bucket has space to store the fingerprint, a bucket (i_1 or i_2) is chosen uniformly at random and the recursive process of cuckoo *eviction* is triggered [4]. Partial-key cuckoo hashing is thus calculated as follows (for fingerprint function $\phi_x = x_f(x)$):

$$\begin{aligned} i_1 &= H(x) \pmod m \\ i_2 &= i_1 \oplus H(\phi_x) \pmod m \end{aligned} \tag{1}$$

Importantly, i_1 and i_2 represent the indices of the buckets, and thus must be translated into the bucket range $[0, m-1]$ to prevent illegal memory references. State-of-the-art implementations [1] use modulo operation for this purpose. When m is a power of two, this modulo operation is reduced to a bit-wise "and", which eventually allows for higher operations throughput on modern hardware.

Additionally, most probabilistic filters are designed with a targeted false positive rate $\epsilon > 0$. For example, given the false positive rate ϵ, a space-optimized Bloom filter uses $\frac{1}{\ln 2}\log_2(\frac{1}{\epsilon})$ bits per item. On the other hand, cuckoo filters use f bits to fingerprint each item, and the minimal fingerprint size for a given false positive rate ϵ and bucket size b is $f = \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil$ [4]. Thus, for the fixed ϵ we can choose b and f to make cuckoo filters that are smaller than corresponding Bloom filters. This property of cuckoo filters makes them a plausible alternative to Bloom filters when we need to minimize memory consumption.

3 CAUSES OF FALSE NEGATIVES

The partial-key alternative bucket calculation given in (1) is false negatives free only when m is a power of two. Otherwise, when m is not constrained to powers of two, we have the following observations:

Observation 1. *There exist cuckoo filters that permit evictions to an incorrect bucket i, such that $i \neq i_1$, and $i \neq i_2$.*

Observation 2. *An element x that is evicted to an incorrect bucket is a false negative when there is no previously inserted x' such that:*

$$(\phi_{x'} = \phi_x) \text{ and } (H(x') = H(x) \pmod m). \tag{2}$$

When the condition from (2) holds, we observe not only the false positives (dictated by parameter ϵ), but also observe false *negatives*. For traditional applications of probabilistic filters, a significant amount of false *negatives* is unacceptable.

Graf and Lemire [5] have recently proposed an alternative version of partial-key cuckoo hashing to replace (1) and avoid modulo operation. However, their version results in a twofold degradation in the throughput of the *Insert* operation [5], which may be unacceptable in traditional network applications.

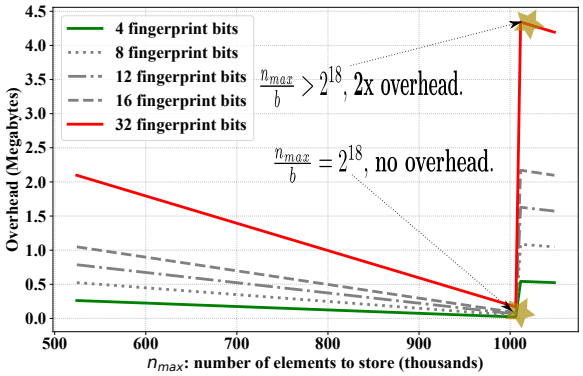

Figure 3: Space overhead as a function of n_{max} for $b = 4$ (referential implementation [1]).

4 EXPERIMENTAL EVALUATION

We measure the sizes of the cuckoo filters [1] for various capacities in Fig. 3. When m is a power of two, the maximal overhead is $f(n_{max} - 2)$ bits and occurs when $\lceil n_{max}/b \rceil = 2^k + 1$, for some integer k (the peak on Fig. 3 occurs when $n_{max}/4 = 2^{18} + 1$).

Next, we compare cuckoo filters [1] (with m constrained to powers of two) with Bloom filters from Bitcoin Core [2]. For a fixed $\epsilon \leq 3\%$, Bloom filters are smaller than cuckoo for most filter capacities (See Fig. 1). These results suggest that one need to relax the constraints on m to achieve the maximal space efficiency of cuckoo filters.

To evaluate the empirical false *negatives* rate, we offer an alternative cuckoo filter implementation based on the state-of-the-art [1]. The only modification is that we permit all possible values m (*i.e.,* not just powers of 2). Fig. 1 depicts the false *negatives* rate as a function of the desired filter capacity n_{max}. We see that for the load factors lower than 80%, the false *negatives* rate remains close to 1%. However, the cuckoo filters that are 95% full lead to up to a 10% false *negatives* rate. The false *negatives* are caused by cuckoo evictions to the incorrect buckets during inserts, when the condition from *Observation* 2 holds. Interestingly, the false *negative* rate of 10% is even higher than the 3% false positive rate given by ϵ. In security applications such as IP packet filtering [3], a false *negative* could allow a malicious packet to be processed as benign, therefore possibly compromising the security of the system.

ACKNOWLEDGMENTS

This research was supported in part by NSF under grant CCF-1563753.

REFERENCES

[1] Efficient Computing at Carnegie Mellon. 2019. Cuckoo Filter. https://github.com/efficient/cuckoofilter. (2019).
[2] Bitcoin Core. 2020. Bitcoin. https://github.com/bitcoin/bitcoin/blob/master/src/bloom.h. (2020).
[3] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. 2003. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.* IEEE, 44–51.
[4] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. *CoNEXT '14* (2014).
[5] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *Journal of Experimental Algorithmics (JEA)* 25, 1 (2020), 1–16.