A More Space-Efficient Implementation of Cuckoo Filter

Mingxun Zhou 1600012706
Yunfan Zhang 1600012710

1 Introduction

A *filter* is a data structure that is used to decide whether an element is a member of a set. One of the most popular filter is the Cuckoo filter, which provides high lookup performance with small false positive probablity and supports dynamically insertion and deletion of elements.

But a limitation of the standard Cuckoo filter, described in 2014 by Fan, Andersen, Kaminsky and Mitzenmacher [1], is that the number of its buckets must be a power of two, because it uses XOR operation to find the alternate bucket. If the number of all elements in the set is given, using standard Cuckoo filter may cause large space consumption in the worst case.

So we propose a new technique using ADD/SUB instead of XOR to find the alternate bucket. It can be applied in cases where the number of buckets is even. In the rest of this article, we will review the Cuckoo filter in section 2 and show the details of our technique in section 3, then compare the performance of these two implementations in section 4.

2 Preliminaries

Cuckoo filter is a compact variant of Cuckoo hash table [2] that stores much shorter *fingerprints* instead of keys. There are some false positives due to the collision of fingerprints and the fingerprint length determines the false positive rate.

A challenge of storing only fingerprints is that we cannot relocate the existing fingerprint, bacause the alternate bucket is determined by two hash values according to the original key. To overcome this problem, a technique called *partial-key cuckoo hashing* is applied. For an item x with fingerprint p, the two candidate buckets are derived as follows:

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= T_{hash(p)}(h_1(x)) \end{aligned} \tag{1}$$

where $T_d(i)$ is a function with one parameter d that meets the following conditions:

- **closure**: $T_d(i)$ must be an index of one bucket

Code : https://github.com/wuwuz/Compact-Data-Structure-Project

- **self-inverse**: $T_d(T_d(i)) \equiv i$
- **aggregation-resistance**: $T_{d_1}(i)$ and $T_{d_2}(i)$ should be different when $d_1 \neq d_2$

If bucket i stores fingerprint p, we can find its alternate bucket by calculating $j = T_{hash(p)}(i)$ no matter whether i is $h_1(x)$ or $h_2(x)$.

The notations used in the following context are listed in Table 1.

ε	false positive rate	b	number of entries per bucket
f	fingerprint length in bits	m	number of buckets
α	load factor ($0 \leq \alpha \leq 1$)	n	number of items
U	bucket index space	q	number of lookups

Table 1: Notations

3 Partial-Key Cuckoo Hashing

3.1 Algorithm Description

The main idea of this technique is to pair up all the buckets according to parameter d and use $T_d(\cdot)$ to find the conjugate bucket. In standard Cuckoo filter, bitwise exclusive-or operation is used, i.e., $T_d(i) = i \oplus d$. The bucket index space U must be formed as $\{0, 1, \cdots, 2^l - 1\}$ in order to satisfy the closure property, thus m must be a power of two.

Now consider another way of pairing, each bucket i is matched with bucket $i \pm d$ as Figure 1 shows. More precisely, if $\lfloor i/d \rfloor$ is even then bucket i is matched with $i + d$, otherwise matched with $i - d$. Some buckets may remain unpaired, and their indices must be consecutive. We call that such bucket is in the *overflow interval*.

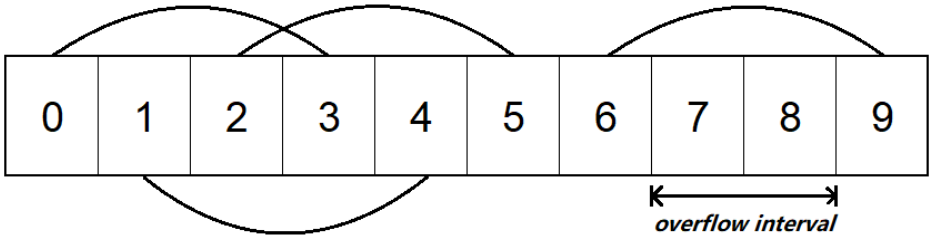

Figure 1: Bucket pairs when $m = 10$ and $d = 3$

Assume bucket i is in the overflow interval of length L, then any other overflow index j needs to meet three conditions: (1) $\lfloor j/d \rfloor = \lfloor i/d \rfloor$; (2) $j < m$; (3) $j + d \geq m$. Hence we can determine the overflow interval with a simple mathematical derivation.

Pick all these buckets out and rehash d into the range from 1 to $L - 1$. Now the problem is reduced to pairing up buckets in the overflow interval, which is a subproblem. We only need to run the above algorithm recusively.

But it remains an issue of aggregation. Consider $m = 2z + 2$ and then any divisor d of z will make a pair between bucket $m - 1$ and $m - 2$. In this case, items with different fingerprints may

correspond to the same two candidate buckets and it will reduce the load factor of Cuckoo filter. To prevent aggregation, some *bias* relevant to the fingerprint is added to the index before paring.

Algorithm 1: Alternate(idx, p)

$bias \leftarrow p \cdot \lfloor n/2^f \rfloor$
$idx \leftarrow (idx + bias) \bmod m$ `/* add some bias to the index */`
$d \leftarrow hash_{[1,m-1]}(p)$ `/* hash the fingerprint */`
$len \leftarrow m,\ trans \leftarrow 0$
while ***true*** **do**
  if $\lfloor idx/d \rfloor$ *is even* **then**
    if $idx + d < len$ **then**
      $result \leftarrow idx + d$
      break
    else
      `/*` idx `is in the overflow interval */`
      $lb \leftarrow \max\{d \cdot \lfloor idx/d \rfloor, len - d\}$
      $rb \leftarrow \min\{d \cdot \lfloor idx/d \rfloor + d - 1, len - 1\}$
      $len \leftarrow rb - lb + 1$
      $d \leftarrow hash_{[1,len-1]}(d)$ `/* rehash` d `into` $[1, len-1]$ `*/`
      $trans \leftarrow trans + lb$
      $idx \leftarrow idx - lb$ `/* translate` $[lb, rb]$ `to` $[0, len-1]$ `*/`
  else
    $result \leftarrow idx - d$
    break
$result \leftarrow (result + trans) \bmod m$
$result \leftarrow (result - bias) \bmod m$
return $result$

3.2 Complexity Analysis

In standard Cuckoo filter, finding the alternate bucket needs only one bitwise operation, while in our technique needs to run a recursive algorithm. In the following, we consider hash function as random variable uniformly distributed on range space, and analyze the expected running time of this algorithm.

Worst-case running time Let W_m be the worst-case expected recursion depth when there are m unpaired buckets. The length of the overflow interval is at most d, while d is uniform random from 1 to $m-1$. In the worst case, the target index is always in the overflow interval until $m = 2$, so we can get a recurrence relation:

$$W_m = 1 + \frac{1}{m-1}(W_1 + W_2 + \cdots + W_{m-1}), \quad W_1 = 0 \tag{2}$$

By solving the above recurrence relation as follows:

$$\begin{aligned}(m-1)W_m &= (m-1) + W_1 + W_2 + \cdots W_{m-2} + W_{m-1} \\ (m-2)W_{m-1} &= (m-2) + W_1 + W_2 + \cdots W_{m-2} \\ (m-1)W_m - (m-2)W_{m-1} &= 1 + W_{m-1} \\ W_m = W_{m-1} + \frac{1}{m-1} &= 1 + \frac{1}{2} + \cdots + \frac{1}{m-1} = \Theta(\ln m)\end{aligned}$$

we know that the worst-case expected time complexity of this algorithm is $O(\log m)$.

Average-case running time Let A_m be the average-case expected recursion depth. If the length of the overflow interval is L, the relative frequency of a target index to be in the overflow interval is approximately L/m. Therefore, A_m can be estimated as follows:

$$A_m = 1 + \frac{1}{m-1}\left(\frac{1}{m}A_1 + \frac{2}{m}A_2 + \cdots + \frac{m-1}{m}A_{m-1}\right), \quad A_1 = 0 \tag{3}$$

By means of mathematical induction, it is easily proven that $A_m \leq 2$, i.e., the average-case expected time complexity of this algorithm is $O(1)$.

4 Evaluation

4.1 Achieved False Positive Rate

First we evaluate the false positive rate that two types of Cuckoo filter achieve. In each run, the filters are configured to have $m = 2^{18}$ buckets each consisting of $b = 4$ entries, and $q = 10^6$ randomly generated keys are used to measure false positives after the filters are filled.

fingerprint length	6	8	10	12	14	16
Standard Cuckoo	11.341%	2.947%	0.758%	0.209%	0.0673%	0.0328%
Add/Sub Cuckoo	11.344%	2.953%	0.763%	0.204%	0.0667%	0.0325%

Table 2: Achieved false positive rate with different length of fingerprint. Each entry is the average of 10 runs.

4.2 Load Factor

Here we evaluate the load factor of two implementations. In each run, the filters are configured to have $b = 4$ entries per bucket and each fingerprint has $f = 8$ bits.

4.3 Required Number of Buckets

Next we will evaluate the required number of buckets to store a given number of items completely. Each filter is still configured with $b = 4$ and $f = 8$.

number of buckets	2^{10}	1536	2^{12}	6144	2^{14}	40000	2^{16}	142856
Standard Cuckoo	0.976	—	0.968	—	0.962	—	0.955	—
Add/Sub Cuckoo	0.973	0.972	0.965	0.967	0.959	0.958	0.956	0.954
	2^{18}	323072	2^{19}	10^6	2^{20}	$2 \cdot 10^6$	2^{21}	
	0.950	—	0.949	—	0.947	—	0.943	
	0.951	0.949	0.951	0.948	0.947	0.945	0.943	

Table 3: Load factor with different number of buckets. Each entry is the average of 25 runs.

number of items		1000	15570	30000	2^{16}	$2 \cdot 10^5$	10^6
Standard Cuckoo	number of buckets	512	4096	8192	2^{15}	2^{16}	2^{19}
	load factor	0.488	0.950	0.916	0.500	0.763	0.477
Add/Sub Cuckoo	number of buckets	262	4064	7876	17162	52614	264154
	load factor	0.954	0.958	0.952	0.955	0.950	0.946

Table 4: Required number of buckets when stored different number of items.

4.4 Insert/Lookup Performance

Last we evaluate the insert and lookup performance. Filters are configured to store 8-bit fingerprints and have 2 or 4 entries per bucket.

Initially the filter is empty and then items are placed until seeing an insert failure. The construction time can in some way reflect the insert performance. After the filter is filled, randomly generated keys are used to measure the lookup performance.

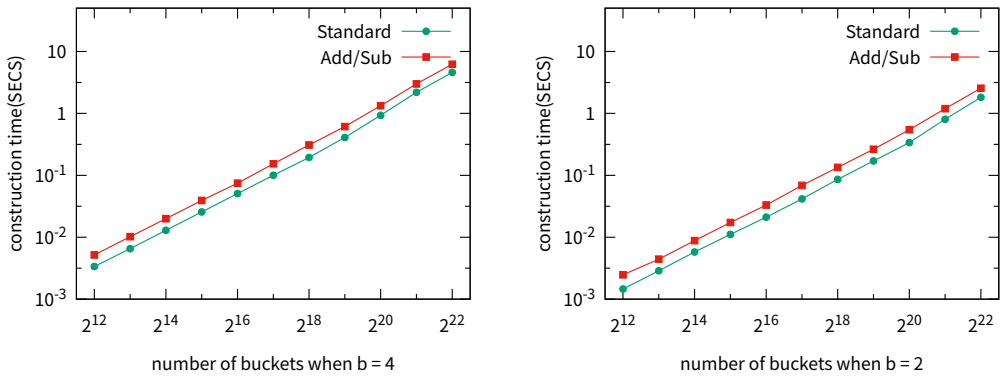

Figure 2: Construction time with different number of buckets. Each point is the average of 10 runs.

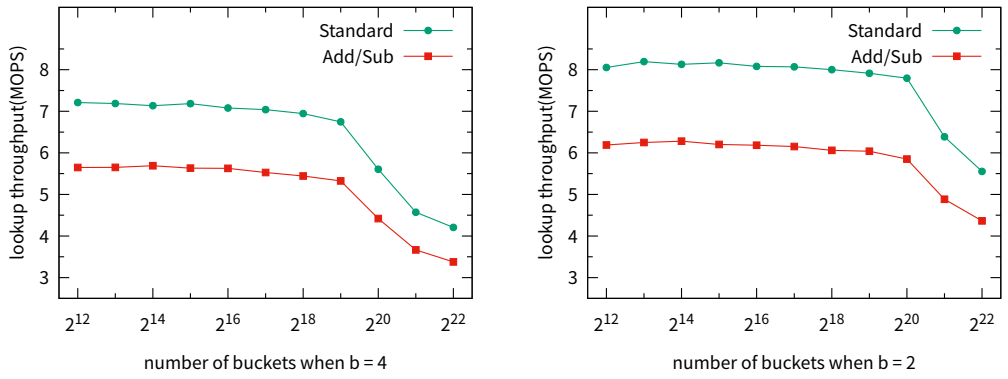

Figure 3: Lookup throughput with different number of buckets. Each point is the average of 10 runs.

5 Conclusion

The achived false positive rate and load factor of these two implementations of Cuckoo filter are nearly the same. The standard Cuckoo filter inserts about 45% and lookups about 30% faster than the Add/Sub Cuckoo filter. But as Table 4 shows, it performs very poorly when the given number of items is near a power of two, while the load factor of Add/Sub Cuckoo filter is always about 95%. So in some special case where the number of items to be stored is configured, the Add/Sub Cuckoo filter will be a better implementation which is more space-efficient.

6 Contributor

- Mingxun Zhou : Come up with the main idea and implement two basic types of Cuckoo Filter.
- Yunfan Zhang : Implement an optimized version Filter, write the evaluation programs and finish the report.

References

[1] B. Fan, D. G. Andersen, M. Kaminsky and M. D. Mitzenmacher, Cuckoo filter: Practically better than Bloom. In *Proc. 10th ACM Int. Conf. Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.

[2] Rasmus Pagh, Cuckoo Hashing for Undergraduates, March 2006. URL http://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf