Bloom-Cuckoo 过滤器：使用集成的 Bloom 过滤器加速 Cuckoo 过滤器的插入过程

李嘉杰

（同济大学 软件工程学院，上海 200000）

摘要： Cuckoo 过滤器（Cuckoo filters）是 Bloom 过滤器（Bloon filters）的一个替代方案，Cuckoo 过滤器不仅提供了 Bloom 过滤器所没有的删除功能，同时也降低了误报率（false positive rate）。但 Cuckoo 过滤器的缺点是它的插入操作相对复杂，尤其是当过滤器占用率（occupancy）较高时每一次插入操作需要进行大量的内存访问。因此，插入操作的高复杂度往往会限制 Cuckoo 过滤器的应用，尤其是在许多需要快速更新、反应的互联网应用中。

因此在本文中，我使用集成 Bloom 过滤器的方法对 Cuckoo 过滤器进行了扩展，来优化 Cuckoo 过滤器的插入过程。在本文所提出的 Bloom-Cuckoo 过滤器中，并不需要额外的内存访问来维护原有的检索和删除操作。此外，因为集成的 Bloom 过滤器的空间占用可以忽略不计，且 Bloom-Cuckoo 过滤器本身所要使用的内存宽度可以根据存储桶（bucket）的宽度进行灵活调整，所以它很适合在硬件上进行部署。通过性能验证，Bloom-Cuckoo 过滤器可以将最坏情况下的插入耗时减少至朴素 Cuckoo 过滤器的 1/10，与检索平均耗时接近。此外，在面对高并发的情况时，Bloom-Cuckoo 过滤器也比朴素的 Cuckoo 过滤器有着更好的表现，它可以在保持更低误报率的同时支持数以百计的连续插入访问。

关键词：近似成员过滤器；Cuckoo 过滤器；Bloom 过滤器

Bloom-Cuckoo Filter: Using Integrated Bloom Filter speeding up the insertion of the Cuckoo filter

Jiajie LI

(Tongji University,Shanghai 200000, China)

Abstract: The Cuckoo filter is an alternative to the Bloom filter, which not only provides the deletionfunction not found in the Bloom filter, but also reduces the false positive rate. However, the disadvantage of the Cuckoo filter is that its insertion is relatively complex, especially when the filter occupancy is high and each insert requires a lot of memory access. As a result, the high complexity of insertion operations tends to limit the application of Cuckoo filters, especially in many Internet applications that require rapid updates and reactions. So in this article, I extended the Cuckoo filter using the integrated Bloom filter method to optimize the procedure for inserting the Cuckoo filter. In the Bloom-Cuckoo filter presented in this article, additional memory access is not required to maintain the original retrieval and deletion operations. In addition, because the space footprint of the integrated Bloom filter is negligible, and because the memory width to be used by the Bloom-Cuckoo filter itself can be flexibly adjusted to the width of the bucket, it is well suited for hardware deployment. With performance verification, the Bloom-Cuckoo filter can reduce the worst-case insertion time to 1/10 of the simple Cuckoo filter, close to the average retrieval time. In addition, the Bloom-Cuckoo filter performs better than the simple Cuckoo filter in high concurrency situations, supporting hundreds of continuous insertion visits while maintaining a lower false sun rate.

Key words: Approximate member filter; Cuckoo filter; Bloom filter

近似成员过滤器广泛用于网络应用程序中，以简化和加速数据处理[1]。例如，Bloom 过滤器[2]通常用于数据包分类[3]和提高安全性。Cuckoo 过滤器[4]是最近提出的近似成员过滤器数据结构，作为 Bloom 过滤器的替代方法引起了人们的关注。Cuckoo 过滤器支持删除元素，并且在某些情况下，

使用相同大小的内存时，Cuckoo 过滤器的误报率低于 Bloom 过滤器。结合在最坏的情况下仅需要两次内存访问即可完成检查这一事实，这使其在某些应用程序场景下有特别的用武之地。特别是，最近已经提出了将 Cuckoo 过滤器，用于以改善深度数据包检查[5]，最长前缀匹配[6]，流量监控的安全性[7]和身份验证[8]等领域。

尽管最初是在通用处理器上的软件中实现的[4]，但是 Cuckoo 过滤器（作为 cuckoo 哈希）也适合于硬件实现[9]，[10]。例如，可以将过滤器存储在双端口内存中，以便可以同时访问两个存储桶，并在单个时钟周期内完成查找。在硬件实现中，可以调整存储器的宽度以适合存储桶的大小。例如，在[11]中提出了 Morton 过滤器，以通过将多个存储桶分组到适合高速缓存行的块中来优化 Cuckoo 过滤器，并通过将大多数元素放在第一个存储桶中来使用块来提高性能。不幸的是，这些优化不适用于硬件实现，因为它们需要使用更宽的存储器，从而增加了功耗，存储器带宽和互连复杂性。

Cuckoo 过滤器也有缺点。其中之一是插入可能需要替换之前插入的元素，就像 cuckoo 哈希[12]一样。这使得插入过程比 Bloom 过滤器更为复杂。在高占用率下，插入可能需要数百次内存访问。这限制了 Cuckoo 过滤器在需要支持快速插入的应用程序中的使用，例如在 SDN 网络中。

在 SDN 网络中执行规则更新的常用方法是分批汇总更改[13]，目的是优化转发规则的安装并最小化更新网络状态的时间。例如，在[13]中考虑了成百上千条规则更新的插入批次。结果，给定网络节点的两次更新之间的时间往往很长，因为所有更新都集中在一个批次中，而不是发送独立的更新。但是，如[14]中所述，批量插入成为 OpenFlow 硬件交换机的一项挑战。

在硬件实现中，存储可以用来临时存储 cuckoo 哈希建议的等待插入的元素[15]。但是，这意味着必须在检查期间比较存储桶中的所有元素，从而增加了额外的复杂性，并且在实际配置中将存储柜的大小限制为几个元素。因此，对于 Cuckoo 过滤器而言，高速支持大量连续插入仍然是未解决的问题。

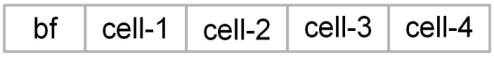

图 1　Bloom-Cuckoo 过滤器 bucket 示意图

在本文中，我提出在 Cuckoo 过滤器中集成 Bloom 过滤器，以提高其处理插入的能力。 主要思想是，在需要时，可以在 Bloom 过滤器上进行无法在 Cuckoo 过滤器中找到空单元的插入，从而减少了插入时间。 所提出的 Bloom-Cuckoo 过滤器已被实施和测试，表明它可以有效地处理大量连续插入。 最坏情况下插入时间的减少可以达到 10 倍，而高占用率的平均时间可以减少 10 倍以上。 不需要检查操作就可以进行额外的内存访问，这对于硬件实现对查找延迟的影响可以忽略不计，并且支持删除所有插入的元素，从而实现了这一目标。 因此，提出的 Bloom-Cuckoo 过滤器可能是一种支持在硬件实现的 Cuckoo 过滤器中支持大量连续插入的有效方法。

1 Cuckoo 哈希

Cuckoo 哈希是一种开放式寻址，哈希表中的每个非空单元格都包含一个键或键-值对。哈希函数用于确定每个键的位置，并且可以通过检查表的该单元格来找到其在表中的存在（或与之关联的值）。但是，开放式寻址会遇到冲突，当多个键映射到同一单元时会发生冲突。布谷鸟哈希的基本思想是通过使用两个哈希函数而不是仅一个哈希函数来解决冲突。这为每个键在哈希表中提供了两个可能的位置。在该算法的一种常用变体中，哈希表被分为两个大小相等的较小表，并且每个哈希函数都提供了这两个表之一的索引。这两个哈希函数还可能在单个表中提供索引。

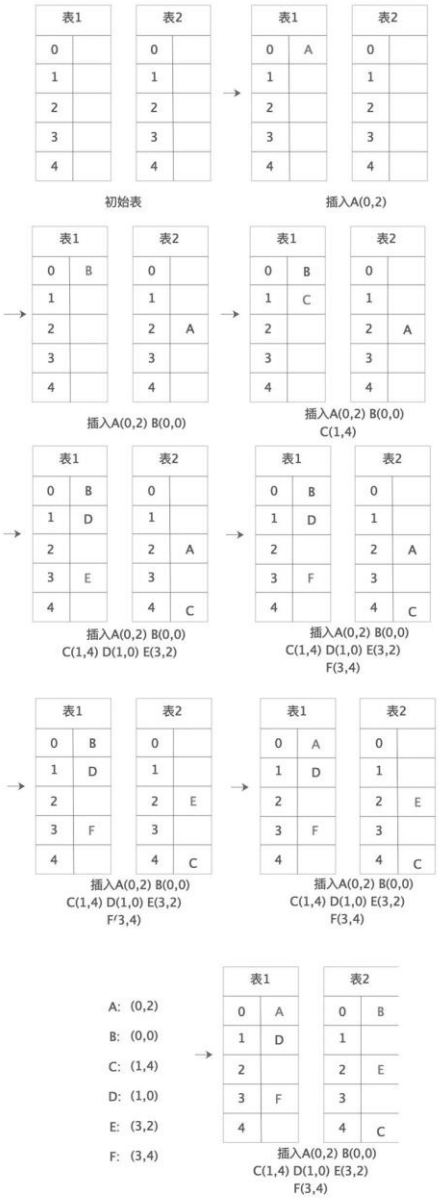

图 2 Cuckoo 哈希示意图

查找仅需要检查哈希表中的两个位置，在最坏的情况下，这需要花费恒定的时间。这与许多其他哈希表算法形成对比，后者在进行查询的时间上可能没有恒定的最坏情况范围。与常数线性探测等其他方案相比，在恒定的最坏情况下，也可以通过对包含密钥的单元进行消隐来执行删除操作。

当插入新密钥，并且其两个单元格之一为空时，可以将其放置在该单元格中。但是，当两个单元都已满时，有必要将其他键移到其第二位置（或移回其第一位置），以便为新键腾出空间。使用贪婪算法：将新密钥插入其两个可能的位置之一，即“踢出”，即，替换可能已经驻留在该位置的任何密钥。然后，将这个移位的键插入其替代位置，再次踢出可能驻留在此处的任何键。该过程以相同的方式继续进行，直到找到空位置为止，从而完成了算法。

2 Cuckoo 过滤器

Cuckoo 过滤器使用 Cuckoo 哈希中的核心思想[4]，[12]存储插入元素的指纹。使用哈希函数 $g\ x$ 计算指纹，使得 $fp_x = g\ x$ 并可以存储在两个存储区中，分别由

$$a_1 = h_1\ x$$

$$a_2 = h_1\ x\ \ xor\ h_2\ fp_x$$

给出。通常，每个存储桶都有四个单元，因此一个存储桶中最多可以存储四个指纹。事实证明，这种配置的方法几乎可以达到全部的占用率[4]。

查找操作仅需要计算 fp_x，a_1，a_2，并随之访问桶 a_1，a_2 查找是否有对应匹配 fp_x 的指纹，如果是则返回真，否则返回假。由于在存储桶中只存储了指纹 fp_x 而不是完整的元素 x，所以可能会出现误报的情况。即当 y 存储在 a_1，a_2 中， 并且 $fp_y = fp_x$ 时。误报布谷鸟过滤器的速率可以近似为

$$fpr_{cf} \simeq \frac{8 \cdot o}{2^f}$$

其中，o 是滤波器占用率，f 是指纹 fp_x 中的位数[4]。 过滤器的存储桶数通常用 m 表示，由于每个存储桶有四个单元格，因此过滤器最多可以存 $4•m$ 个指纹。

删除与插入类似，但是一旦找到匹配的指纹，就可以将其删除。插入相比之下更为复杂，因为当尝试插入元素时，我们可能会发现两个存储桶 a_1，a_2 都已满。在这种情况下，我们将随机地选择 a_1，a_2 中的某一个指纹，并由我们尝试插入的元素的指纹代替。然后尝试对另一个存储桶中的已删除元素进行新的插入，也就是说，如果已将其从 a_1 中删

除，则尝试将其插入 a_2 中，反之亦然。

因此，不难发现当过滤器占用较高时，插入元素所需的此类移动次数可能会很大，从而限制了 Cuckoo 过滤器持久性快速更新的能力。

3 集成了 Bloom 过滤器的 Cuckoo 过滤器

为了将 Bloom 过滤器添加到 Cuckoo 过滤器中，我提出在过滤器的每个存储桶中附加一个额外的位 *bf*。如图 1 所示，每一个存储桶将会同时拥有可以存储四个指纹的四个单元和一个 Bloom 过滤器位组成。对于一个给定元素 *x*， Bloom 过滤器会检查其对应的 a1 和 a2 的两个位置的 *bf* 位。这意味着 Cuckoo 过滤器和 Bloom 过滤器都只需要通过访问这两个桶便可以完成校验。因此，集成的 Bloom 过滤器不需要额外的内存访问。

所以，对于检查操作而言，如果 fp_x 与存储在存储桶 a_1，a_2 中的任何指纹之间匹配，或者如果 a_1 和 a_2 中的位 *bf* 都为 1，则检查操作返回真值。这一集成的操作仅需要将检查 *bf* 位（硬件实现中为使用一个二输入与门）的逻辑添加到朴素的 Cuckoo 过滤器检查过程中即可。

与常见的 Bloom 过滤器不同，为了避免发生冲突，在我的 Bloom-Cuckoo 过滤器中，集成的 Bloomn 过滤器中的插入操作是受到限制的。

对于插入操作而言，在我提出的 Bloom-Cuckoo 过滤器中，我们可以将待插入的新元素 *x* 插入 Cuckoo 鸟过滤器或 Bloom 过滤器中。此外，已经插入到 Cuckoo 过滤器中的元素 *y* 也可以插入到 集成的 Bloom 过滤器中，因为我们可以确定其对应的 a1，a2 存储段。但是，已经插入到 Bloom 过滤器中的元素 *z* 不能再移回 Cuckoo 过滤器。

接下来实现 Bloom-Cuckoo 过滤器中的插入和删除操作。

对于 Bloom 过滤器的删除操作而言，常见的问题往往是无法从 Bloom 过滤器删除元素。这意味着无法从 Bloom-Cuckoo 过滤器中删除某些元素，从而消除了朴素 Cuckoo 过滤器特有的删除功能。为了避免这个问题，在我提出的 Bloom-Cuckoo 过滤器中，只有在存储段 a_1，a_2 上的两个 *bf* 位在插入前都为零的情况下，才能将元素 *x* 插入 Bloom 过滤器中。这意味着可以避免在 Bloom 过滤器上发生冲突。因此，相应的可以通过将桶 a_1，a_2 上的 *bf* 位设置回 0 来删除元素。这种策略的缺点是，在给定的时间内我们也许无法在 Bloom 过滤器中插入元素 *x* 。但是，从性能评估部分的实验结果中可以看到，这对插入时间的影响有限。因为在发生这种情况时，可以先执行 Cuckoo 移动，然后尝试在移走的元素 *y* 上插入之前尝试在 Bloom 过滤器上插入的元素。

Algorithm 1 Removal of an Element x

```
1: Compute h1(x),fpx,h2(fpx),a1,a2
2: Access buckets a1,a2
3: if fpx found then
4:     Remove fpx
5:     return success
6: end if
7: if both bf bits of a1,a2 are one then
8:     Set both bf bits to zero
9:     return success
10: end if
11: return fail
```

Algorithm 1 Bloom-Cuckoo 过滤器中的删除操作

当需要删除 Bloom-Cuckoo 过滤器中的某一个元素时，我们首先检查是否有与其匹配的指纹，如果有，我们将其删除。如果不存在，则继续检查两个 *bf* 位；如果它们均为 1，则将它们设置为 0，以从集成的 Bloom 过滤器中删除该元素。在 Algorithm 1 中对此进行了详细的伪代码说明，该算法仅应用于先前插入过滤器中的元素。

因为需要在过滤器中进行大量的实时插入，所以提出的 Bloom-Cuckoo 过滤器的主要目标是减少 Cuckoo 过滤器的插入时间。

Algorithm 2 Insertion of a New Element x

```
Compute h1(x),fpx,h2(fpx),a1,a2
Set i = 1
Access buckets a1,a2
if Empty cells then
    Select one empty cell randomly and insert fpx there
    return success
end if
if (i > t) and both bf bits of a1,a2 are zero then
    Set both bf bits to one
    return success
end if
Select randomly a bucket a between a1,a2 and a fingerprint
    stored in it fpy
Remove fpy and place fpx on its place
Compute be as a xor h2(fpy) and assign fpe to be fpy
for i ← 2 to t do
    Access bucket be
    if Empty cell then
        Insert fpe there
        return success
    end if
    Select randomly a bucket a between a and be
    Select randomly a fingerprint fpz stored in a
    Remove fpz and place fpe on its place
    Compute be as a xor h2(fpz)
    Assign fpe to be fpz
end for
for i ← t+1 to maxiter do
    Access bucket be
    if Empty cell then
        Insert fpe there
        return success
    end if
    if both bf bits of a,be are zero then
        Set both bf bits to one
        return success
    end if
    Select randomly a bucket a between a and be
    Select randomly a fingerprint fpz stored in a
    Remove fpz and place fpe on its place
    Compute be as a xor h2(fpz)
    Assign fpe to be fpz
end for
return fail
```

Algorithm 2 Bloom-Cuckoo 过滤器中的插入操作

提出的算法背后的思想是先尝试迭代进行 *t* 次的朴素 Cuckoo 过滤器的插入过程，然后在尝试使用集成的 Bloom 过滤器的插入过程作为第二选项。 这意味着从第 *t+1* 次迭代开始，由于存在两种插入的可能性，成功率会进一步增加。将 *t* 设置为较小的值将更有效地减少插入时间，但会在集成的 Bloom 过滤器上引起更多的插入，这会降低误报率，并且还会使 Bloom 过滤器对后续插入的效率降低。另一方面，使用较大的 *t* 值会最大程度地减少 Bloom 过滤器上的插入，但在减少插入时间方面效果不佳。因此，*t* 值的选取需要根据具体情况权衡，选择合适值。

集成的 Bloom 过滤器的副作用是会在一定程度上增加误报率，因为现在查找操作可能是由 Cuckoo 过滤器或集成的 Bloom 过滤器来完成。因此，当两个过滤器的假阳性率均远小于 1 时，Bloom-Cuckoo 过滤器的假阳性率可以近似为 $fpr_{cfbf} \simeq fpr_{bf} + fpr_{cf}$。为了量化 Bloom 过滤器的影响，我们假设在 Bloom 过滤器上插入了 *n* 个元素，然后 *bf* 位为 1 的位数为 *2 n*（为避免插入冲突）。因此， Bloom 滤波器的误报率约为 $fpr_{bf} \simeq (\frac{2 \cdot n}{m})^2$ 。根据 $m \cdot 2^{\frac{1-f}{2}} \cdot \sqrt{o} \gg n$ ，当 $fpr_{cf} \gg fpr_{bf}$时，Bloom 过滤器的影响效果会很弱。由于仅当 Bloom-Cuckoo 过滤器在高占用率下工作时才需要使用 Bloom 滤波器，因此 *o* 可以近似为 1，并且对于高占用率滤波器，条件可以进一步简化为$m \cdot 2^{\frac{1-f}{2}} \gg n$。因此，当 *m* 大而 *f* 小时，该方案可以支持 *n* 个元素的短时大量插入。以上分析的大型过滤器和适度的误报率恰好是许多网络应用中的实际情况。

最后，作为进一步的优化，在使用集成的 Bloom 过滤器处理完短时间内大量连续插入之后，从 Bloom 过滤器中删除这些元素并将其插入到 Cuckoo 过滤器中是对性能有益的。这将能够减少 Bloom 过滤器的占用率并用于应对将来的插入突发，而不会影响误报率。为此，可以使用一个表来存储插入到 Bloom 过滤器中的元素。在那种情况下，当没有查找或插入时，该表中的元素可以插入到 Cuckoo 过滤器中，并从 Bloom 过滤器中删除。其中，对于每个元素 *q*，我们只需要在表中存储 *h1*（*q*）和 fp_q。

Bloom-Cuckoo 过滤器引入的内存开销是：添加到每个存储桶的 *bf* 位以及刚刚讨论的 Bloom 过滤器中插入的元素表。但是因为该表在查找期间

无需检查，因此可以存储在速度较慢的存储器中，例如外部存储器，所以对于它来说，开销可以忽略不计。因此，唯一的片上存储器开销是每个存储桶的一位 *bf*。对于实际情况最糟的 8 位指纹，额外开销将小于 3%，这在许多设计中都是可以接受的。

4 性能评估

为评估所提出方法在减少插入所需迭代次数方面的有效性，Bloom-Cuckoo 过滤器和 朴素的 Cuckoo 过滤器已在 C ++ 中实现。为此，我们模拟了一个表，其中有 $m = 32768$ 个存储桶，指纹为 f=12 位。在考虑插入失败的情况的同时，最多要尝试 1000 次 Cuckoo 迭代。

测试程序会插入元素，直到过滤器达到给定的占用率为止。朴素 Cuckoo 过滤器的选定占用值为 $o = 0.92$、0.94、0.95。为了进行公平的对比，对提出的 Bloom-Cuckoo 过滤器的占用率进行了调整，以使每个元素的存储位数与 Cuckoo 过滤器中的位数相同。这导致我们的 Bloom-Cuckoo 过滤器的占用值为 $o = 0.9392,0.9596,0.9698$。占用量较大意味着 Bloom-Cuckoo 过滤器中存储了更多的元素，以补偿 Bloom-Cuckoo 过滤器中为 bf 位添加的额外内存，从而每个元素的内存位保持不变。

在最初将表格填充到所需的占用率之后，将删除随机元素并插入新元素的替换操作，以将存储在过滤器中的元素数量增加一倍，以使其达到稳定状态。此时，使用修改的插入过程执行 $n = 256$ 次突发插入，该修改的插入过程使用 Bloom 过滤器处理不同的 t 值。记录最大和平均插入时间，以及突发末尾 Bloom 过滤器中存储的插入次数。对于每一种不同的配置，将运行 100 次试验，并且给出的结果对应于所有试验中的最大值或平均值。

性能评估结果如图 3、4、5 所示。可以看出，所提出的 Bloom-Cuckoo 过滤器能够减少最大和平均迭代次数。当占用率接近 1 时，最大迭代次数可以在 Cuckoo 滤镜上采用较大的值。

在图 3 中可以清楚地看到这一点，当占用率为 0.94 或更高时，可以看到超过 100 的值。相反，当 $t = 10$ 且占用率为 0.9698 时，出现的 Bloom-Cuckoo 过滤器的最坏情况最大值仅为 19。因此，Bloom-Cuckoo 过滤器可以将最坏情况的插入时间降低到更低的值。平均插入时间也会发生同样的情况，如图 4 所示。在这种情况下，Bloom-Cuckoo 过滤器所获得的值与 t 有着明显的相关性。这是可以预期的，因为建议的插入算法在迭代 t 之前不会尝试在 Bloom 过滤器上插入。对于 $t = 0$ 的值（这样，布鲁姆过滤器可用于所有迭代），即使占用率为 0.9698，平均迭代次数也非常接近一（1.022）。这意味着如果需要，Bloom-Cuckoo 过滤器可以在突发期间保持每次迭代的插入速率为 1。最后，应该注意的是，对于较短的插入突发，观察到了迭代次数的类似减少，并且在所有情况下，模拟的误报性率均与理论近似值相符。图 5 显示了 256 次插入突发后在 Bloom 过滤器上插入的元素的平均数量。可以看出，即使当 $t = 0$ 时，也不会将所有元素都插入到 Bloom 过滤器中。因此，Bloom-Cuckoo 过滤器可以维持一个比上一节中计算的 n 大的突发。还可以看出，插入 Bloom 过滤器中的元素数量随着 t 的增加以及占用率的降低而减少。因此，如果需要，t 的值可用于限制在 Bloom 过滤器中插入的元素数量。

设计第二组性能评估主要目标是分析突发插入量对 Bloom-Cuckoo 过滤器性能的影响。为此，将 Cuckoo 过滤器的占用率设置为 $o = 0.95$，将 Bloom-Cuckoo 过滤器的占用率设置为 $o = 0.9698$，以使两个过滤器每个元素的存储位数相同。重复测试过程，但这次的突发大小为 $n = 128$、256、384、512、640、768、896、1024，迭代次数 $t = 0$、5、10。

显示了迭代次数的结果在图 6、7 中可以看到，

Cuckoo 滤波器的平均迭代次数和最大迭代次数都随突发大小而增加，而 Bloom-Cuckoo 过滤器的值保持稳定。这可以解释为在 Cuckoo 过滤器中，插入时间在很大程度上取决于占用率，并且突发大小越大，过滤器将达到的占用率值就越大。相反，对于 Bloom-Cuckoo 过滤器，插入时间主要取决于 *t* 的值，因为一旦超过 t，可以将元素放置在 Bloom 过滤器上，并且将在第一次迭代中成功。图 8 显示了 Cuckoo 过滤器和Bloom-Cuckoo 过滤器插入突发后的误报率。可以看出，Bloom-Cuckoo 过滤器的误报率随着脉冲串大小的增加而增加，并且当脉冲串大小大于 256 时开始产生明显的影响，如第三部分中的分析所预测的那样。突发大小为 1024 时，误报性率增加不到三倍。这表明，如果在将元素从 Bloom 过滤器移至 Cuckoo 过滤器之前，误报率的下降，Bloom-Cuckoo 过滤器可以支持更大数量的突发插入。还可以看出，随着 *t* 的增加，下降较小，因为在那种情况下， Bloom 过滤器中放置的元素更少。

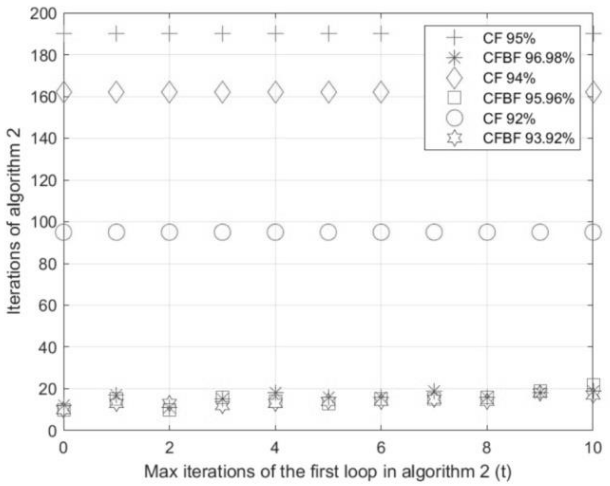

图 3

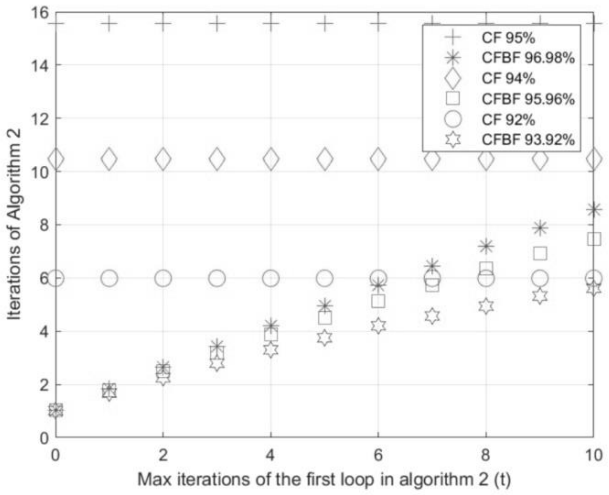

图 4

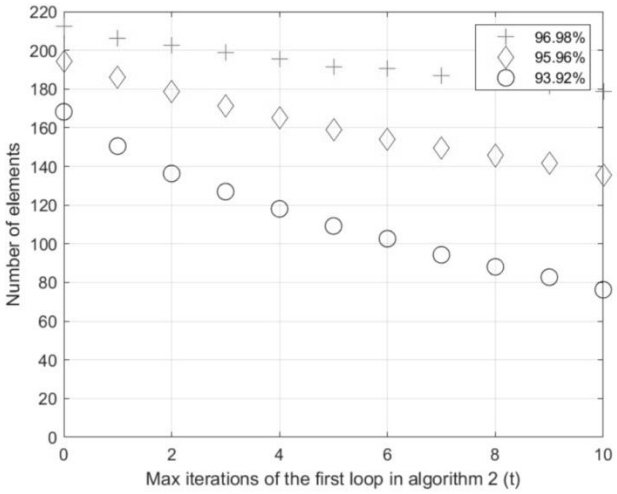

图 5

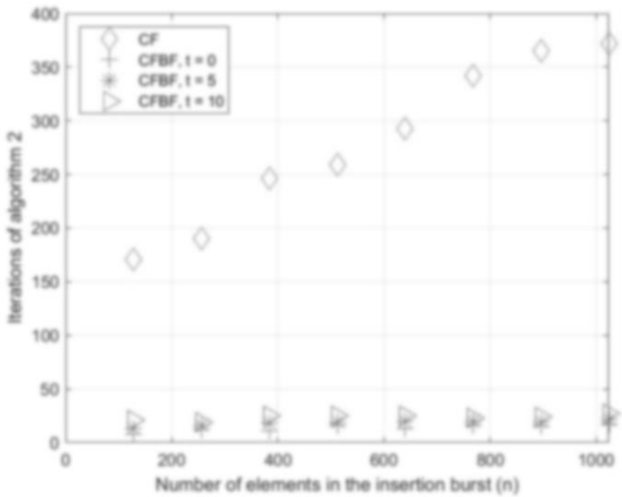

图 6

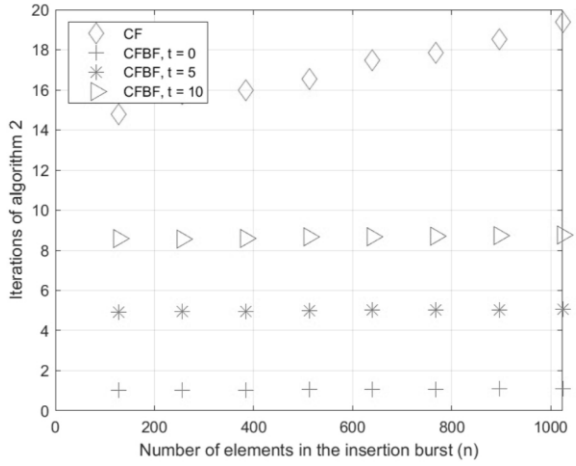

图 7

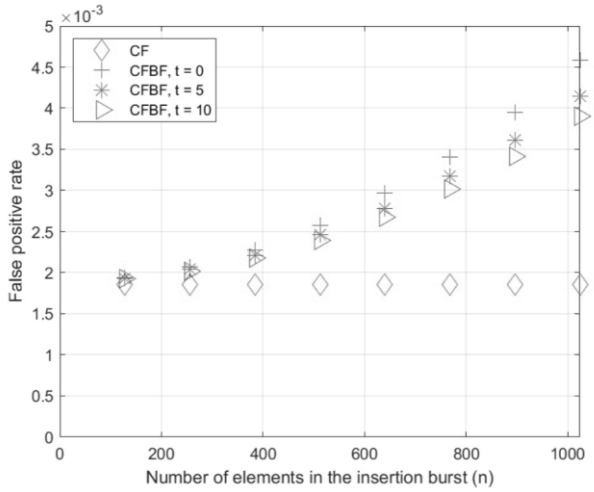

图 8

5 总结

Cuckoo 过滤器的局限性之一是当过滤器以高占用率运行时执行插入操作会非常复杂，所需的时间会很长，在本文中，我提出了 Bloom-Cuckoo 过滤器来解决此问题。提出的 Bloom-Cuckoo 过滤器保留了原始的 Cuckoo 过滤器的主要功能，支持删除，并且不需要其他内存访问即可进行查找。Bloom-Cuckoo 过滤器已通过 C++实现并进行了性能测试。结果表明，它可以大大减少平均和最坏情况下的插入时间。这使其可以用于需要对过滤器中的大量元素进行快速更新的应用场景。

为了进一步提升 Bloom-Cuckoo 过滤器短时突发插入的能力，值得探索的是如何进一步挖掘 Bloom 过滤器的低误报率。一种选择是在 Bloom 过滤器中使用指纹代替单个 *bf* 位以减少其误报率[16]。此外，通过设计更多的 bf 位，使得 Bloom-Cuckoo 过滤器可以避免在 Bloom 过滤器上发生碰撞，从而使指纹更有效。例如，对 Bloom 过滤器指纹使用每个存储桶两个位将使误报率降低 9 倍。有关使用指纹 Bloom 过滤器的详细分析留待以后的工作继续推进。

参考文献：

[1] A. Kirsch, M. Mitzenmacher, and G. Varghese, “Hash-based techniques for high-speed packet processing,” in Algorithms for Next Generation Networks. London, U.K.: Springer, 2010, pp. 181–218.

[2] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” Internet Math., vol. 1, no. 4, pp. 485–509, 2004.

[3] P. Gupta and N. McKeown, “Algorithms for packet classification,” IEEE Netw., vol. 15, no. 2, pp. 24–32, Mar. 2001.

[4] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo filter: Practically better than bloom,” in Proc. CoNext, 2014, pp. 75–88.

[5] M. Al-Hisnawi and M. Ahmadi, “QCF for deep packet inspection,” IET Netw., vol. 7, no. 5, pp. 346–352, Sep. 2018.

[6] M. Kwon, P. Reviriego, and S. Pontarelli, “A length-aware cuckoo filter for faster IP lookup,” in Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS), Apr. 2016, pp. 1071–1072.

[7] J. Grashöfer, F. Jacob, and H. Hartenstein, “Towards application of cuckoo filters in network security monitoring,” in Proc. 14th Int. Conf. Netw. Service Manage. (CNSM), Nov. 2018, pp. 373–377.

[8] J. Cui, J. Zhang, H. Zhong, and Y. Xu, “SPACF: A secure privacy- preserving authentication scheme for VANET with cuckoo filter,” IEEE Trans. Veh. Technol., vol. 66, no. 11, pp. 10283–10295, Nov. 2017.

[9] G. Levy, S. Pontarelli, and P. Reviriego, “Flexible packet

matching with single double cuckoo hash," IEEE Commun. Mag., vol. 55, no. 6, pp. 212–217, Jun. 2017.

[10] P. Bosshart et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM), 2013, pp. 99–110.

[11] A. D. Breslow and N. S. Jayasena, "Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity," Proc. VLDB Endowment, vol. 11, no. 9, pp. 1041–1055, 2018.

[12] R. Pagh and F. F. Rodler, "Cuckoo hashing," J. Algorithms, vol. 51, no. 2, pp. 122–144, 2004.

[13] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostic´, "ESPRES: Transparent SDN update scheduling," in Proc. 3rd ACM Workshop Hot Topics Softw. Defined Netw., 2014, pp. 73–78.

[14] M. Kuz´niar, P. Perešíni, and D. Kostic´, "What you need to know about SDN flow tables," in Proc. Int. Conf. Passive and Active Netw. Meas., 2015, pp. 347–359.

[15] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," SIAM J. Comput., vol. 39, no. 4, pp. 1543–1561, 2009.

[16] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Improving counting Bloom filter performance with fingerprints," Inf. Process. Lett., vol. 116, no. 4, pp. 304–309, Apr. 2016.