Improving Cuckoo Hashing with Perfect Hashing

Moulika Chadalavada and Yijie Han
School of Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110, USA.
mc7d8@mail.umkc.edu, hanyij@umkc.edu

Abstract - This paper mainly aims at improving Cuckoo hashing by using Perfect Hashing to store the keys in memory based on frequency. Perfect Hashing is fast and hit ratio is high in Perfect Hashing. Cuckoo Hashing has high memory usage in allocating keys to its memory. So, combining Cuckoo Hashing and Perfect Hashing will increase the keys hit ratio.

Keywords: Tree, Hashing, Cuckoo Hashing, Perfect Hashing, Algorithms

1 Introduction

There are many hashing techniques that aim at storing keys in memory to increase key access efficiency and to make hashing efficient. In network applications packet classification plays very prominent role [2]. One option to increase throughput is to use the algorithms based on hashing [3]. Hash Table or Hash Map is a data structure that is used in implementing structure that can map keys to values. A hash table uses Hash Function to compute index into array, from which the required values are found. The main disadvantage of Hash Tables is that it maps multiple keys to same index thus results in collision in hashing. To handle this issue many hashing techniques are introduced to avoid collisions in allocating memory.

Cuckoo Hashing is one of the hash table schema which provides high memory utilization and constant access time [4]. Cuckoo Hashing mainly aims at reducing collisions and optimizing the throughput. There were many implementations in Cuckoo Hashing such as Serial Implementation, Parallel Implementation, Parallel Pipeline Implementation, Parallel d-Pipeline Implementation [1].

Perfect Hash Function is a hash function which maps distinct element of subset S to set of integers with no collision. However, in perfect hashing the set of keys to be hashed must be provided to create the hash function. In mathematical term, it is total injective function. This hash function is used in implementing lookup table with constant worst-case access time. There are many hash functions that are like Perfect Hashing but the main advantage is that no collision resolution should be implemented in Perfect Hashing. In this paper, we use Perfect Hashing to improve Cuckoo Hashing in terms of memory utilization and allocating memory based on frequency of keys.

2 What is Cuckoo Hashing

Cuckoo hashing uses a number d of hash tables and an element x can be placed in those tables 1, 2, …d in positions h1(x), h2(x), …... where hi(x) is hash functions [1]. The main difference between d-left hashing [5] and Cuckoo hashing is that, in d-left hashing when all positions are occupied the new element cannot be inserted and limits memory usage. But in Cuckoo Hashing the elements in occupied position is moved to their alternative positions to insert new element. There are many implementations of Cuckoo Hashing which aims at increasing throughput [1]. The first case is Serial implementation in which tables are accessed serially and in Parallel implementation, the table is selected in random.

In Pipeline architecture [1], searching access each memory sequentially i.e. when current option moved to memory-2, the second search operation can start accessing memory-1. In this pipeline implementation when one search is successful other memories need not to be accessed. In Parallel d-pipeline [1] each pipeline has different entry point which allows the user to insert an element to any table idle in that cycle. If an element is in first pipeline and match found in Table-1 then in next cycle element will be inserted in Second pipeline to make use of Table-2.

3 What is Perfect Hashing

A Perfect Hash function for a set S is a hash function that maps distinct elements in S to a set of integers, with no collisions. Minimal Perfect Hashing [9] guarantees that n keys will map to 0..n-1 with no collisions at all. Given set of n keys, a static hash table of size m=O(n) can be constructed such that Search takes O(1) time in the worst case. A perfect hash function can be used with limited range of values used for efficient lookup operations, this can be done by placing keys from subset S in lookup table indexed by function’s output. Then one can test whether key present in S, by looking at its cell of table and each lookup takes constant time in worst case.
As discussed above, Perfect hashing is a technique for building hash table with no collisions. This is possible when all the keys are known in advance. Minimal Hashing means the resulting hash table contains only one entry for each known key and no empty slots exists. To insert keys to slots two levels of hash functions are used [9]. First is $H$ (key), hash the key that gets position in intermediate array $G$. The second function, $F(d, key)$ uses extra information of $G$ to find unique position of the key. This scheme always returns value, if we know for sure that the key we are searching for is in the table. Otherwise, it returns bad information.

**How the Intermediate Value can be found in Perfect Hashing?** [9]

1. We keep keys into buckets according to first hash function, $H$ (key).

2. Then we process the buckets largest first and try to place all the keys it contains in an empty slot. If that is unsuccessful, we keep trying with successively larger values of $d$. It sounds like it would take a long time, but it doesn't. Since we try to find the $d$ value for the buckets with the most items early, they are likely to find empty spots. When we get to bucket with just one item, we can simply place them into the next unoccupied spot. [5]

**4 Use of Perfect Hashing to improve Cuckoo Hashing**

**4.1 Allocating Key to Memory**

We show how to use perfect hashing to improve Cuckoo hashing by considering the frequency of keys. We cannot anticipate all possible keys because the set of keys is a huge set. For example, if keys are limited to no more than 20 letters then the set of keys has size $12720$ which is a huge size set. However, we can put known frequently encountered keys into a set $S$ and then map the keys in $S$ to memory modules by using perfect hashing function $f$. Such a perfect hashing function can be obtained in $O(|S|2b)$ time [7], where $b$ is the number of bits to represent a key in $S$. After $f$ is obtained, $f(x)$ for a key $x$ can be computed in constant time [7]. In Cuckoo hashing every key is assumed to have the same priority. Here we analyze the set $S$ of frequently encountered keys and store high frequency keys together in a memory module. Because there are few keys with high frequency and more keys with less frequency we may, say, store keys with frequency above 50% in memory module 0, store keys with frequency 20% to 50% in memory module 1, store keys with frequency 5% to 20% in memory module 2, and store the keys with frequency less than 5% in memory module 3.

The architecture of our scheme is, for an input key $x$, first compute its perfect hash value $f(x)$. According to [7] the value of $f(x)$ in within $\{0, 1, \ldots, |S|\}$, thus, no matter $x$ is in $S$ or not, $f(x)$ will always return a value in $\{0, 1, \ldots, |S|\}$. We then use the value of $f(x)$ to index into a table $T$ that stores the memory module number for $f(x)$ value. Thus, if $x$ is in $S$ then we find the correct memory module that stores $x$. Say $x$ is in $S$ and the memory module for store $x$ is $M_a$. Then $T[f(x)] = a$. After we know memory module $M_a$, we then use a hash function $h$ for $M_b$ to find the location $h(x)$ of $x$ in $M_a$. If $x$ is not in $S$. Then we will first use $f(x)=a$ to find memory module $M_b$. We then use $h(x)$ to locate $x$ in $M_b$. Three situations can happen here. The first situation is that $h(x)=h(y)$ for a $y$ in $S$. Thus $h(x)$ and $h(y)$ collides. Thus, we know that $x$ is a less frequent key. We can then go to the memory module $M_b$ for storing less frequent keys and hash and rehash $x$ there to identify whether $x$ is already in $M_b$ or need to be inserted into $M_a$. The second situation is that no key is at position $h(x)$ or $x$ is stored at $h(x)$ position of $M_a$. This can happen because $f(x) = f(y)$ for $y$ in $S$ and thus $T[f(x)] = T[f(y)] = a$ and therefore we are going to go to the same memory module $M_a$ for both of them. However, $h(x) \neq h(y)$. Thus, if $h(x)$ position is vacant we then store $x$ at $h(x)$ position of $M_a$. If $x$ is already in the $h(x)$ position of $M_a$ then we found $x$ in $M_a$. The third situation is that $h(x)=h(y)$ for a $y$ not in $S$ while $y$ has already been in the position $h(y)$ in $M_a$. In this situation, again $x$ is a less frequent key and we need to go to the memory for less frequent keys to locate $x$.

Also note that $f(x)$ has $|S|$ values and only $|S|$ values correspond to keys in $S$ and the other $|S|>|S|$ values don't correspond to any key $x$ in $S$. Thus these $|S|>|S|$ values for $f(x)$ correspond to less frequency keys. We can set $T[f(x)]$ to memory for storing less frequency keys. In this way frequently occurred keys in $S$ will be identified in constant time. For those keys not in $S$ their hash value may have collision with the keys in $S$. Since these keys are less frequently occurred and therefore we can afford more hashing and rehashing time for them. Where as in perfect hashing the hashing is fast and hit ratio is high. In perfect hashing, all the keys in the subset $S$ is known. Initially hash $f$ needs to be performed on each key which returns the frequency of the key and the memory module for it. Each key is assigned to memory module via the hash table value for it. Based on the frequency of respective key, the keys are stored in memory modules. The keys with highest frequency are stored in Memory-1 and the lower frequency keys are stored in next memory. If there are any non-frequent words, then they can be stored in Separate Memory.

For example, let’s take below famous sentence stated by Fredrick P. Brooks Jr.

“There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.”
From above sentence the frequently occurring word is ‘in’, that has count of 4 and all other words has count of ‘1’. As per our problem statement each keyword is hashed and distinct hash values from \{0, 1, 2, \ldots, n\} is assigned to each word. An index table is maintained to store both frequency and Memory location for respective word. As ‘in’ is more frequently occurred word, it is stored in Memory-1 and other with less frequency are stored in Memory-2. Once the hashing, memory allocation and updating of index table is completed for all words look up for any word in the memory becomes easy.

### 4.2 Adding new Key to Memory

All the keys are known in the set, so once the memory allocation is completed any key can be looked up in memory based on frequency. If a new key which is unknown has to be stored in memory, first hashing is performed. If the key has highest frequency, then it is looked up in memory that stores keys with high frequencies or if it has less frequency then it will be looked up in memory with low frequencies. If the new key is not a frequent key, then it will be stored in memory which stores non-frequent keys.

Because of this mechanism, the hashing, key storage, memory utilization and key look up is performed very efficiently. When compared to Cuckoo Hashing this mechanism is more efficient as hashing is performed fast and cleverly follows memory utilization. To explain this mechanism with example let us consider below table that contains character’s list and its respective hashed frequencies.

<table>
<thead>
<tr>
<th>Character</th>
<th>Hashed Frequency</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.1 %</td>
<td>Memory-2</td>
</tr>
<tr>
<td>b</td>
<td>0.4 %</td>
<td>Memory-2</td>
</tr>
<tr>
<td>c</td>
<td>4 %</td>
<td>Memory-1</td>
</tr>
<tr>
<td>d</td>
<td>5 %</td>
<td>Memory-1</td>
</tr>
<tr>
<td>e</td>
<td>0.3 %</td>
<td>Memory-2</td>
</tr>
</tbody>
</table>

As per above table Character c & d have highest frequencies with 4% and 5% respectively. So, these two characters are stored in Memory-1. Whereas characters a, b, e has less frequencies with 0.1%, 0.4%, 0.3% respectively, therefore these 3 keys are stored in next memory i.e. Memory-2. The index table is maintained that stores hashed value of each character and its respective Memory Location.

### 5 Conclusions

In this paper, we explained how perfect hashing can be used to improve Cuckoo Hashing with frequency of keys. Also explained how this mechanism is used to increase keys hit ratio and to reduce memory usage. Key lookup in memory based on its frequency will be fast and new key insertion to memory also becomes easy with this mechanism.

### 6 References


