IMPROVING CUCKOO HASHING WITH PERFECT HASHING

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

By
MOULIKA CHADALAVADA

B. Tech, SASTRA University - Thanjavur, India, 2013

Kansas City, Missouri
2017

IMPROVING CUCKOO HASHING WITH PERFECT HASHING

Moulika Chadalavada, Candidate for the Master of Science Degree

University of Missouri – Kansas City, 2017

ABSTRACT

In computer science, the data structure is a systematic way of organizing data such that it can be used efficiently. There are many hashing techniques that aim at storing keys in memory to increase key access efficiency and to make hashing efficient. One option to increase throughput is to use the algorithms based on hashing. Cuckoo Hashing is one among the techniques which provide high memory usage in constant access time. Cuckoo Hashing, in turn, uses many implementations among which Parallel-d-Pipeline is more efficient. Perfect Hashing maps distinct elements to set of integers without any collision. Perfect Hashing is fast and hit ratio is high. There are many other hashing techniques like Perfect Hashing but the reason we choose perfect hashing as it doesn't require collision resolution mechanism. Cuckoo Hashing has high memory usage in allocating keys to its memory. So, we are combining Cuckoo Hashing and Perfect Hashing to increase the keys hit ratio and memory utilization.

Keywords: Cuckoo Hashing, Memory Utilization, Perfect Hashing, Collision Resolution, Hashing

APPROVAL PAGE

The faculty listed below, appointed by the Dean of School of Computing and Engineering, have examined a thesis titled “Improving Cuckoo Hashing with Perfect Hashing” presented by Moulika Chadalavada, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yijie Han, Ph.D., Chair
School of Computing and Engineering

Baek-Young Choi, Ph.D.
Department of Computer Science Electrical Engineering

Mohammad Kuhail, Ph.D.
Department of Computer Science Electrical Engineering

CONTENTS

ILLUSTRATIONS

Figure Page

TABLES

Table Page

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank following people who have directly or indirectly helped me in academic achievements. Firstly, I would like to thank Dr. Yijie Han for the constant and endearing support which has helped me in fulfilling my thesis. He has provided me with an opportunity to realize my potential in the field of my thesis. His encouragement and inputs were elements of vital guidance in my thesis. He has been a constant source of motivation and challenged me with deadlines, that have contributed to me acquiring inspiration and ideology. His expertise and innovative insights have been phenomenal in completing my thesis.

I sincerely thank Dr. Baek-Young Choi and Dr. Mohammad Kuhail for accepting to be a part of my thesis committee and making time for me from their busy schedule. I would like to thank the University of Missouri- Kansas City for providing me with an opportunity to continue my research and supporting me in this regard.

I would like to dedicate my thesis to my parents who constantly inspired me to pursue higher studies. I would like to thank my family who stood behind me all these years during my degree. Finally, I would like to thank all my teachers, educational administrators, present, and past and all who helped me achieve this academic goal.

CHAPTER 1

INTRODUCTION

In computer science, the data structure is a systematic way of organizing data such that it can be used efficiently. A good algorithm generally comes with the set of good data structures which allows designed algorithms to efficiently manipulate data. Hash Table is one of the data structure that maps keys to values using the hash function. Hashing is the mechanism that is used for storing and indexing a large amount of data using keys.

There are many hashing techniques that aim at storing keys in memory to increase key access efficiency and to make hashing efficient. One option to increase throughput is to use the algorithms based on hashing [2]. Hash Table or Hash Map is used in developing a structure that maps keys to values. Hash Table uses a hash function to compute an index into the array, from which the required values are found. The main disadvantage of Hash Tables is that it maps multiple keys to the same index thus results in the collision in hashing [9]. To handle this issue many hashing techniques are introduced to avoid collisions in allocating memory.

Cuckoo Hashing is one of the hash table schemas which provides high memory utilization and constant access time [6]. In this mechanism, every non-empty cell contains a key-value pair. It is a form of open addressing in which every non-empty cell in hash table contains a key-value pair. But open addressing may lead to the collision, this is where Cuckoo Hashing plays its role in resolving collisions. Cuckoo Hashing mainly aims at reducing collisions which might occur when more than one key is mapped to the same cell. It is also used in optimizing the throughput. Cuckoo Hashing uses two hash functions instead of one to reduce collisions. There were many implementations in Cuckoo Hashing such as Serial

Implementation, Parallel Implementation, Parallel Pipeline Implementation, Parallel d-Pipeline Implementation [9].

I studied research paper "Parallel d-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput" by S. Pontarelli, P. Reviriego, and J.A. Maestro. In this paper new Cuckoo Hashing implementation called parallel, d-pipeline has been proposed. In this scheme, tables are accessed parallel such that throughput is increased and is also applicable for applications with high speed.

Perfect Hash Function maps distinct element of a set 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 a 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.

My research basically involves in finding the solution to improve Cuckoo Hashing using Perfect Hashing mechanism in terms of memory utilization and allocating memory based on the frequency of keys.

CHAPTER 2

CUCKOO HASHING AND ITS IMPLEMENTATION

In this chapter, we review paper "Parallel d-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput" by S. Pontarelli, P. Reviriego, and J.A. Maestro.

2.1 Overview

Cuckoo Hashing is one of the hash table schemas which provides high memory utilization and constant access time [6]. In this mechanism, every non-empty cell contains a key-value pair. It is a form of open addressing in which every non-empty cell in hash table contains a key-value pair. But open addressing may lead to the collision, this is where Cuckoo Hashing plays its role in resolving collisions. Cuckoo hashing is used for resolving hash collisions for hash function values in a table [9]. It somewhere derives the features of cuckoo species, where the chick pushes young ones out from nest when it hatches. Similarly, in Cuckoo hashing, while inserting new key into the hash table we can push the older key to the table in a different location.

To avoid collision instead using one hash function Cuckoo Hashing mechanism uses two hash functions. Because of which for each key two locations are provided in the hash table. It takes constant time in the worst case as the lookup of key requires inspection of two locations in the hash table. In Cuckoo Hashing mechanism inserting a new key first look if one of its two cells are empty or full, when empty then the key will be placed in the available cell. But when both the cells are full, the other keys will be moved to their respective second locations to insert the new key.

An element x can be placed in hash tables from 1, 2 …d in positions $h_1(x)$, $h_2(x)$… where $h_i(x)$'s are hash functions [9]. The main difference between d-left hashing [1] and Cuckoo hashing is that in d-left hashing new elements cannot be inserted when all positions are occupied because of which memory usage is limited [9]. But in Cuckoo Hashing to insert a new element, the elements at occupied positions are moved to alternative positions. This is implemented recursively to increase the success rate of inserting an element.

2.2 Example of Cuckoo Hashing

In this section let's consider an example to explain Cuckoo Hashing more efficiently. As discussed above each key has two hash functions to avoid collisions, so let's take a table that contains keys and its respective hash functions.

Table 1: Cuckoo Hashing Example - Keys and Positions

	25	55	58	80	105	72	110	8	41	74
H_1(key)	9	6	9	9	1	1	6	3	3	6
H_2(key)	1	4	4	6	9	6	9	0	3	3

As shown below the keys will be inserted into the hash table. Now first let's insert 25 at its possible position i.e. $H_1(25) = 9$ and next similarly 55 at position 6.

Table 2: Cuckoo Hashing Example – Inserting Keys in Hash Tables

	0	1	2	3	4	5	6	7	8	9	10
Table1	-	-	-	-	-	-	55	-	-	25	-
Table2	-	-	-	-	-	-	-	-	-	-	-

Next key 58 has to be inserted at position $H_1(58) = 9$, but as 25 is already in 9, 25 will be moved to its alternative position $H_2(25) = 1$

Table 3: Cuckoo Hashing Example – Moving Key to Secondary Position

	0	1	2	3	4	5	6	7	8	9	10
Table1	-	-	-	-	-	-	55	-	-	58	-
Table2	-	25	-	-	-	-	-	-	-	-	-

Next to insert 80 we have $H_1(75) = 9$, but 58 is already present at 9 so we place 80 in table 1 and 58 in table 2 at its secondary location H2(58) = 4

Table 4: Cuckoo Hashing Example – Moving Key to Secondary Position2

	0	1	2	3	4	5	6	7	8	9	10
Table1	-	-	-	-	-	-	55	-	-	75	-

Table2	-	25	-	-	58	-	-	-	-	-	-

Similarly, all the other keys are inserted in the hash table as shown below.

Table 5: Cuckoo Hashing Example – Inserting All Keys in respective positions

	0	1	2	3	4	5	6	7	8	9	10
Table1	-	105	-	41	-	-	55	-	-	75	-
Table2	8	25	-	74	58	-	72	-	-	110	-

2.3 Different Cuckoo Hashing Implementations

There are many implementations of Cuckoo Hashing which aims at increasing throughput [9]. In the Serial implementation, tables are accessed serially whereas in Parallel implementation tables are selected in random. In Pipeline architecture [9], 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 have to be accessed. In Parallel d-pipeline [9] each pipeline has different entry point which allows the user to insert an element into any table idle in that cycle.

While inserting an element there might is several free positions among $h_1(x)$, $h_2(x)…h_d(x)$. In such situation, only two algorithms would be required to select the position for inserting element. The first algorithm is to insert an element in the first immediate table that is free. The second option is to insert the element in a table that is free and is selected

randomly. In the first scenario, the table will be accessed serially whereas in second case tables are accessed in parallel. In serial table access, worst case time occurs when an element is not found, or it got inserted in the last table. Whereas the average accesses depend on occupancy of the table and is also lower.

In parallel access, there are d memories and each table is stored in different memories and we can perform search operation parallel accessing all the memories. Here, every search operation requires ‘d' accesses and is also completed in a single memory cycle. We can pipeline search operations to reduce the number of memory access. The performance of pipeline searches might be same as parallel but in pipeline architecture when once the search is successfully completed there is no need of accessing rest of memories.

Parallel Pipeline functionality is implemented in Fig.1 in which the element is searched in table-1, if the element is not found then it starts searching in table-2, table-3 and so on. When the element is found then immediately it returns output and doesn't perform any other searches.

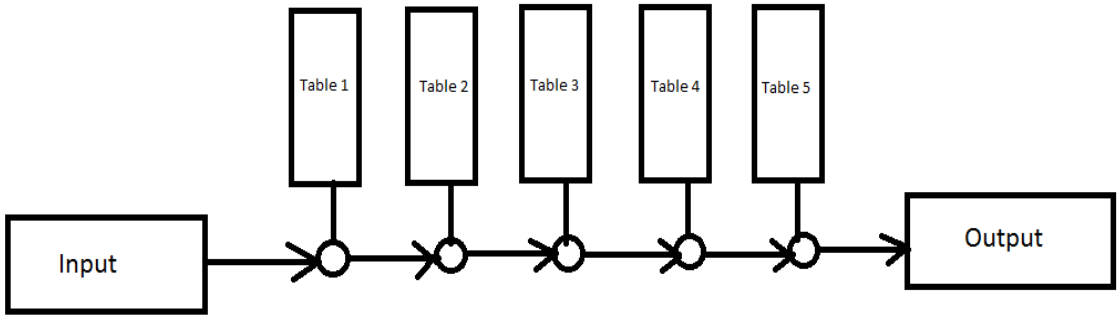

Figure 1: Parallel Pipeline Implementation

2.4 Implementation of Parallel-d-Pipeline

As per Fig.1, in pipeline approach, it has constant throughput and will be equal to an element per cycle. Whereas in Parallel-d-Pipeline approach throughput depends on various factors. One among them is matching operations percentage that is successful, this factor is required because if an operation is failed then it requires accessing all the tables. Another factor is that to check if Cuckoo Hashing uses sequential or random insertion.

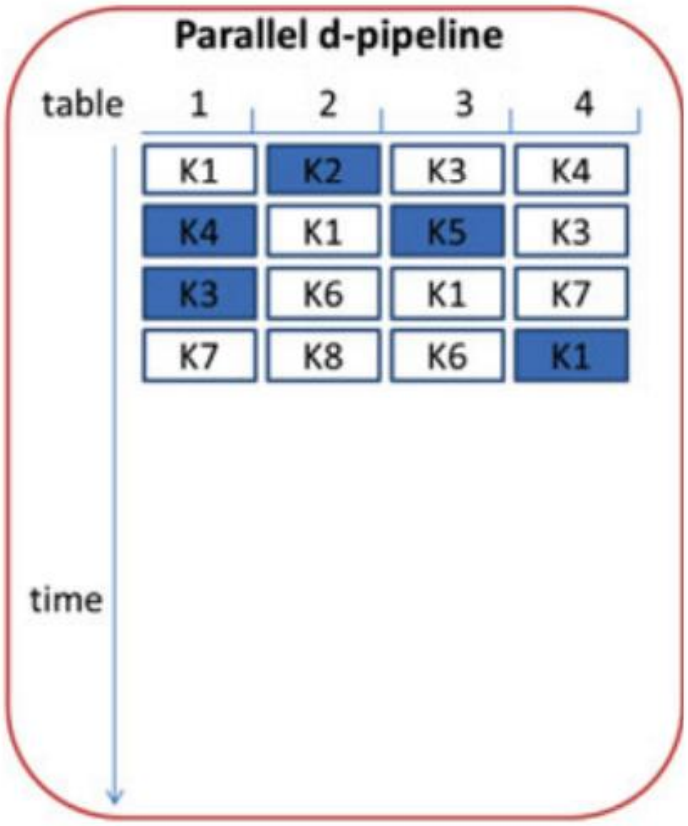

Figure 2: Parallel-d-Pipeline Example

In applications that use Cuckoo Hashing, search operations percentage is close to 100% because if an item is not found in the table then we will add the element to hash table. To evaluate throughput of this new implementation first we need to assume that the occupancy of

the table is 100% and there are m% of searches that are failed. So, in this case, to find an element in d table, the probability of finding it in first access is approximately equal to 1/d, such that in second access it would require (1-1/d) operations and third access needs (1-1/d-1/d) operations and so on. If an element is not in the tables, it would require ‘d' accesses. According to this paper, the number of accesses required in finding a match will be as follows.

$$E\,[\text{No. of Access}] = d \,.\, m + (1 - m) \,.$$

$$(1 + (1 - 1/d) + (1 - 2/d) + (1 - 3/d) + \ldots + (1 - (d-1)/d)$$

$$= d \,.\, m + (1 - m) \,.\, (d + 1) / 2$$

$$= (d + 1) / 2 + m \,.\, (d - 1) / 2$$

Now, if we assume that m is close to zero, and for d = 4 the value of the expected number of accesses would be equal to 2.5. As in this implementation 4 tables are accessed in every cycle, the average throughput per cycle will be approximately equal to 4/2.5 = 1.6. There is overall 60% of increase when compared to simple pipeline implementation. When the occupancy of the table is less than 100% then the probability of finding an element depends on the insertion method that is used.

CHAPTER 3

PERFECT HASHING

3.1 Introduction

In Perfect Hashing hash function maps distinct elements of set S to a set of integers, without collisions. Minimal Perfect Hashing [3] is that which guarantees exactly n keys mapped to 0...n-1 with no collisions at all. Given the set of n keys, a static hash table of size m=O(n) can be created such that Searching takes O(1) time complexity in the worst case. In chained hash table the expected cost of lookup is O(1+α) where α is load factor. In linear probing, the expected cost is O(1). But the expected cost of lookup is not same as expected worst-case cost of lookup. Perfect Hashing lookups take worst case time O(1).

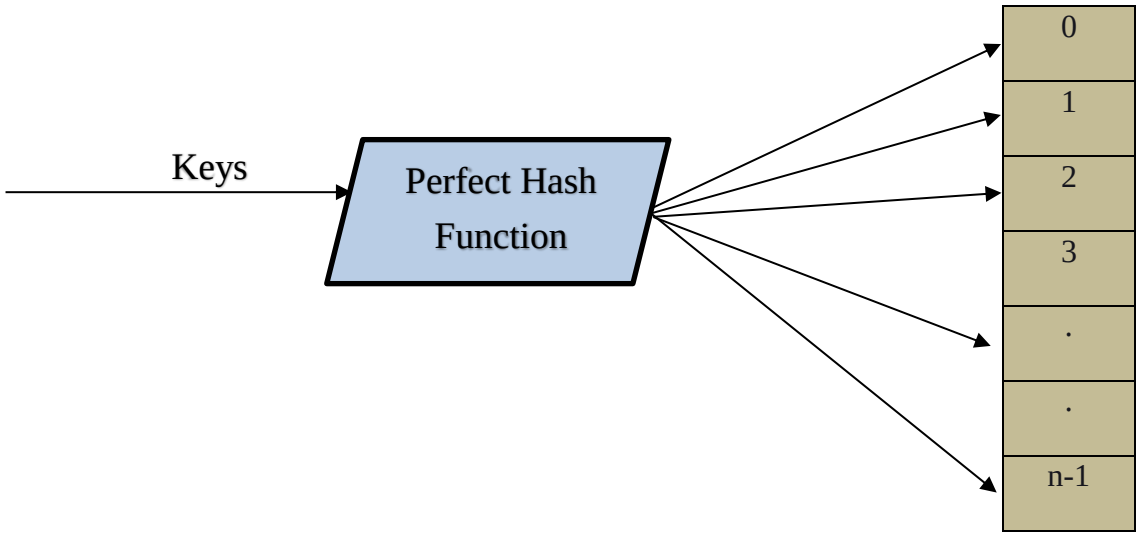

Figure 3: Illustration of Perfect Hashing

A perfect hash function can be operated with a limited range of values that are used for efficient lookup operations. This can be done by placing keys from subset S in a lookup table indexed by function's output. A function is said to be a perfect hash function when the hash function return values that range from 0…n-1. Also, the hash function should be fast, efficient, and easy to compute, and for every key unique value hash to be returned. As discussed above, Perfect hashing is used to build the hash table without collisions which is possible only when all the keys are known in advance.

3.2 Minimal Perfect Hashing

Perfect Hashing make sure that collisions do not occur but might contain empty slots. A minimal perfect hash function is that which maps each key to n consecutive integers such that the hash table will contain one entry for each key and empty slot don't exist as illustrated in Figure 3. Minimal Perfect Hash Functions used for efficient memory usage and for speed elements retrieval from static data sets like URLs in search engines, items in data mining techniques etc. It avoids time and space wastage. As per studies some of the minimal perfect hashing techniques uses 2.1 bits per key [3].

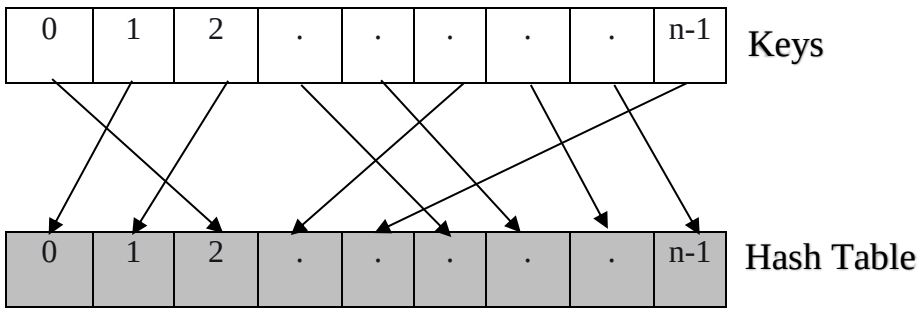

Figure 4: Minimal Perfect Hashing

Two main properties define the minimal perfect hash function, they are Perfect Property and Minimal Property [16].

- ***Perfect Property:*** Lookup in table for an entry would require O(1) time which is maximum a string comparison for element recognition in a static search set.
- ***Minimal Property:*** Memory that is allocated is large enough to store the keyword set.

To insert keys to slots two hash functions are used as shown below [3]. This process will always return a value, only if the key we are looking for is present in the table or else it returns key not found.

1) ***H(key)***, this function hashes the key which returns a position in the intermediate array.
2) ***F(x, key)***, is used to find the position of a key which needs to be unique, where x = H(key).

In perfect hashing, we find intermediate table [3] first by storing keys in buckets according to first hash function, H(key). Next, the first largest bucket is processed and stores all the keys the bucket holds in an empty slot by using F(x=1, Key) [3]. If this step is not successful, then successively we try to process with larger values of x. If the buckets that are returned contains only one item, then we can store them in the slot that is next unoccupied.

Let's take an example to explain Perfect Hash Table Lookup as shown in Figure 4. Here we take two words ‘white' and ‘dog' which hash to the same location using the first hash function **H(white) = H(dog) = 1447**. However, we use the second hash function which is combined with hashed value to place them into different slots.

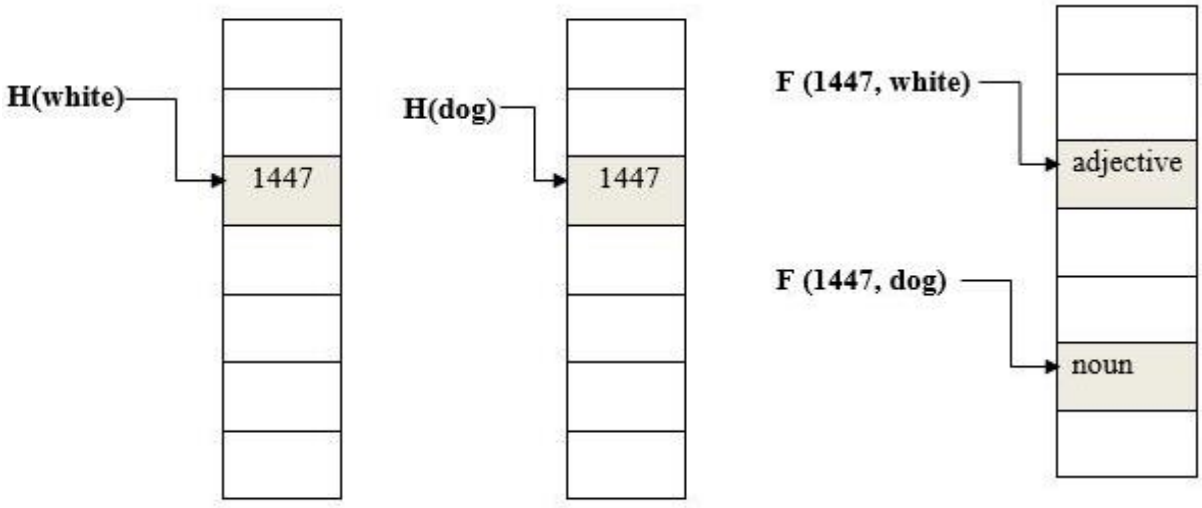

Figure 5: Perfect Hashing Example – Intermediate and Value Table

In minimal perfect hashing, the hash function H, will be order preserving when the keys are in some order such that for keys a_j and a_k, if $j<k$ then $H(a_j) < H(a_k)$ [14]. Here the hash function value will be each key's position in sorting order of all the keys. We can implement this in constant access time by simply using the perfect hash function that stores the position of every key in a lookup table. Generally, the minimal perfect hash function with order-preserving requires Ω (n log n) bits [15].

There are few other variations of perfect hashing which are used in many hashing applications. One among them is ***Non-Minimal Perfect Hash Functions*** (NMPHF) [16], this variation does not satisfy Minimal Property because NMPHF returns set of hash values that are greater than the number of elements in the entire table. Also, Non-Minimal Perfect Hash Functions doesn't satisfy Perfect Property. However, NMPHF is used because it is generated fast than minimal perfect hash functions. Also, Non-Minimal Perfect Hash Functions are executed faster when compared to minimal perfect hash functions.

Another variation of perfect hashing is ***Near-Perfect Hash Function*** [16]. This hash function also doesn't satisfy Perfect Property because it returns hash-values that are not unique. This function has increased generated code execution time and decreased function generation time.

CHAPTER 4

OUR APPROACH

4.1 Introduction

Hashing is a searching technique that is used to find an element in the collection of items using the hash function and hash table. Any algorithm book contains a chapter with hashing and various efficient and elegant hashing techniques. When we talk about hashing we majorly concentrate on collision resolution. We consider collision resolution while hashing due to many reasons. A hash function is said to be a good function based on the application specifically. Even with very best hash function collisions might occur when inserting an element to hash table.

In our previous sections, we discussed two of the best Hashing techniques i.e. Cuckoo Hashing and Perfect Hashing. Cuckoo Hashing is used to resolve collisions that might occur during element insertion in tables. Whereas a Perfect Hashing is that which doesn't contain collisions as it maps distinct elements of set S to a set of integers. In Cuckoo Hashing, to avoid the collision, it requires two hash functions. There might be a possibility that algorithms may enter an infinite loop and may result in the collision. Also, as we are storing in alternate locations the memory usage is more. In Cuckoo Hashing, insertions may run into cycle, which might require multiple rehashes to succeed. Cycles only arise if we revisit the same slot with the same element to insert.

So, in our approach, we chose to combine Cuckoo Hashing and Perfect Hashing mechanism to improve the keys hit ratio and memory utilization. We used Perfect hashing because it doesn't require any collision resolution mechanism and doesn't result in the collision while inserting elements into the hash table. In further sections of this chapter, we show how

to allocate key to memory and how to add a new key to memory by combining Cuckoo Hashing and Perfect Hashing.

4.2 Allocating Key to Memory

In this section, 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 127^{20} 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 a perfect hashing function f. Such a perfect hashing function can be obtained in $O(|S|^2 b)$ time [8], 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 [8].

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 [8] the value of f(x) is within $\{0, 1, \ldots, |S|^2 - 1\}$. Thus, no matter x is in S or not, f(x) will always return a value in $\{0, 1, \ldots, |S|^2\}$. 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_a 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_a. We then use h(x) to locate x in M_a.

The explanation of our approach is explained in-detail with below illustrations and examples.

1) Firstly, to assign key to Memory, the perfect hash function has to be performed on element 'x' which returns frequency of x i.e. f(x). Let's assume value of x='TECHNOLOGY' and its f(x) value is 5.

Figure 6: Illustration of getting frequency of element

2) Once f(x) is calculated, this value has to be indexed to the table to get the Memory module in which f(x) is present.

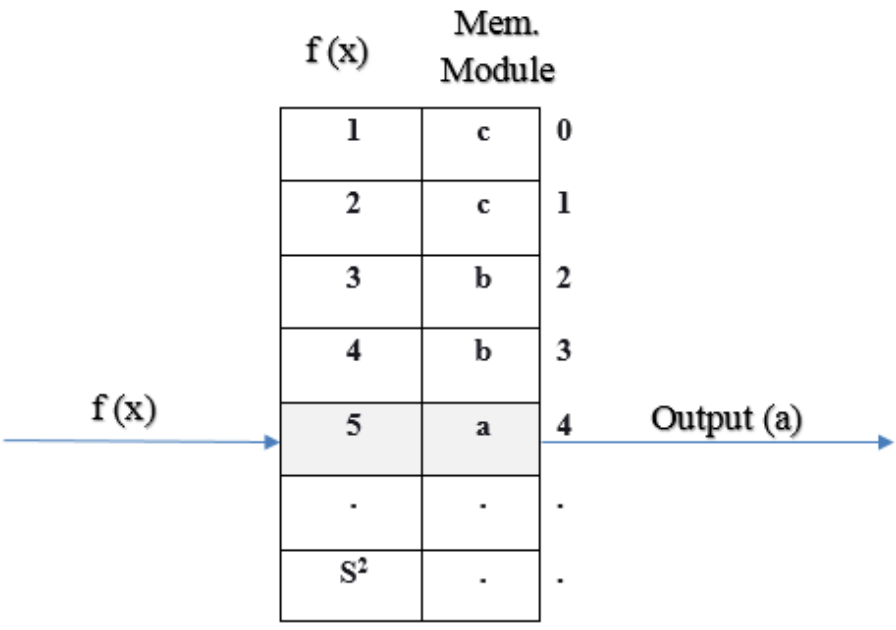

Figure 7: Table Indexing to get Memory Module

As we assumed f(x) = 5, which has high frequency and the Memory Module will be M_a.

3) Further, we use hash function h(x) to get the exact location of element x in memory M_a.

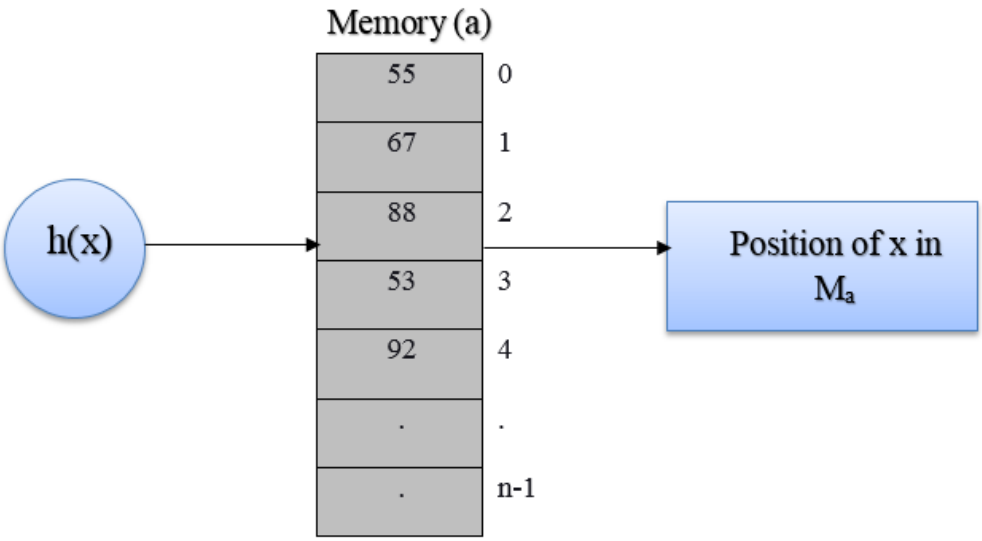

Figure 8: Element lookup in Memory

Three situations can happen here while allocating memory to the element.

i. 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_b.

ii. 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. However, h(x)≠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.

iii. The third situation is when 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 $|S|^2$ values and only |S| values correspond to keys in S and the other $|S|^2$ - |S| values don't correspond to any key x in S. Thus these $|S|^2$ -|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 occur less frequently and therefore we can afford more hashing and rehashing time for them. Whereas in perfect hashing the hashing is fast and hit ratio is high.

In perfect hashing, all the keys in the subset S are 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 the 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.

4.3 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 the 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.

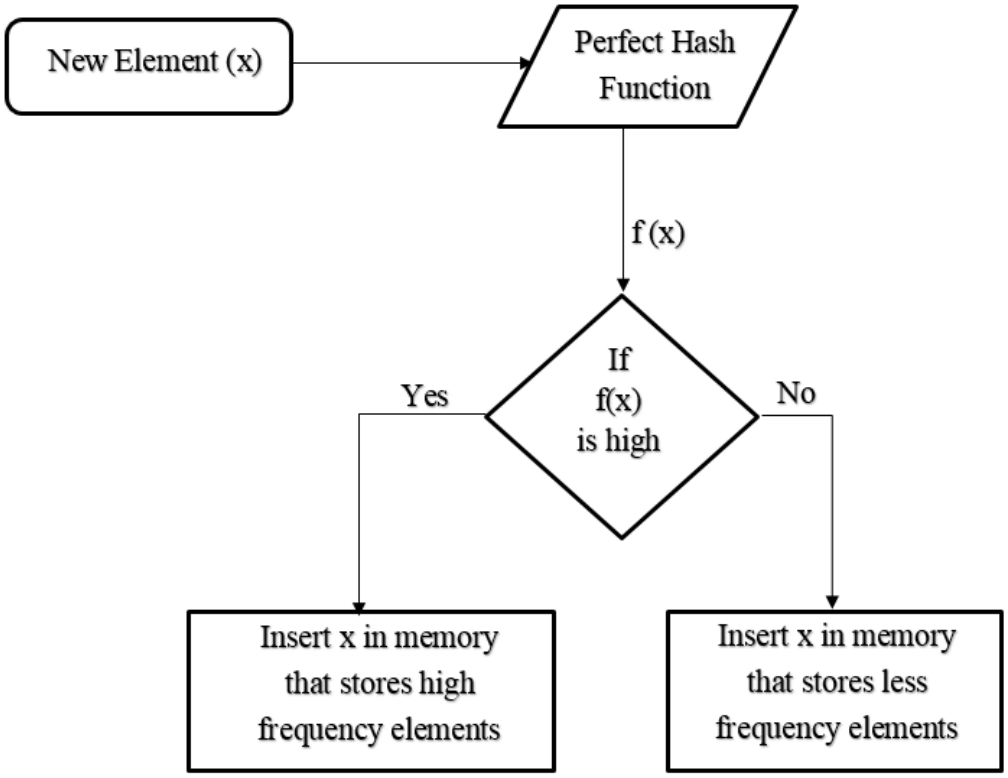

Figure 9: New Element Insertion in Memory

Because of this mechanism, the hashing, key storage, memory utilization and key lookup 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 6: Perfect Hashing Indexing Table Example

Character	Hashed Frequency	Memory
A	0.1%	Memory-2
B	0.4%	Memory-2
C	4%	Memory-1
D	5%	Memory-1
E	0.3%	Memory-2

As per Table-1 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.

CHAPTER 5

IMPLEMENTATION

5.1 Approach for Implementing Algorithm

Firstly, to implement this approach successfully, we need to get frequencies for the set of keys. This should be achieved using Perfect Hashing with which the collisions will not arise, and each element is mapped to unique integer from 0 to n-1. Cichelli's Algorithm is one of the efficient algorithms to achieve minimal perfect hashing. Below steps are followed to achieve this algorithm.

1. Calculate frequency of first and last letters.
2. Order the keys in decreasing order of the sum of the frequencies.
3. Set g-Value to 0 for all first and last characters of words.
4. Assign a 'max' value which indicated highest accepted g-value.
5. Then the hashing of the key is performed using below hash function.

   H(word) = (length(word) + g(firstletter(word)) + g(secondletter(word)) % TSize

   Where TSize : size of table

   length(word) : total length of word

   g(firstletter(word)) : g-Value of first letter

   g(secondletter(word)) : g-Value of second letter
6. If any collision in position occurs for a word, then the g-Value of word's first character is incremented and Step-5 is repeated.
7. Thus, finally all keys are assigned to integers uniquely, thus satisfying the Minimal Perfect Hashing property.

The above algorithm is implemented using Java programming language. Used Collections to store the keys and values for the selected input data. Cichelli's Algorithm is firstly implemented in separate class "PerfectHashFrequencies.java", which takes input and store the keys in String [] array. Further, the above algorithm is implemented in Java as shown below.

Code for Implementing Cichelli's Algorithm

Sample Input: There is no single development in either technology or management technique which by itself promises simplicity

1. **Calculate frequency of first and last characters of each word in the list.**

```
HashMap<Character, Integer> hm = new HashMap<Character,
        Integer>();

for (String key : data) {

      Integer freq;

      // Get the first letter and frequency and place in HashMap
      letter = key.charAt(0);
      freq = hm.get(letter);
      if (freq == null)
            freq = 1;
      else
            freq++;

      hm.put(letter, freq);

      // Get Last Letter and Frequency and place in HashMap
      letter = key.charAt(key.length() - 1);
      freq = hm.get(letter);
      if (freq == null)
            freq = 1;
      else
            freq++;

      hm.put(letter, freq);
}

// Sort the Map in decreasing Order
```

```
        Map<Character, Integer> sortedHM = hm.entrySet().stream()
        .sorted(Collections.reverseOrder(Map.Entry.comparingByValue()))
        .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
        (e1, e2) -> e2, LinkedHashMap::new));

   System.out.println();
   System.out.println("*First & Last Character Occurence of Words*");
   System.out.println(sortedHM);
```

Sample Output:

```
{T=5, E=4, S=4, I=3, Y=3, N=2, O=2, R=2, B=1, D=1, F=1, H=1, M=1, P=1, W=1}
```

2. Sort the keys in decreasing order based on sum of frequencies

```
        // For each Key calculate Sum of first and last characters
        HashMap<String, Integer> wordValueMap = new HashMap<String,
        Integer>();

        for (String key : data) {
          int wordValue = sortedHM.get(key.charAt(0)) +
                          sortedHM.get(key.charAt(key.length() - 1));

           wordValueMap.put(key, wordValue);

        }

          // Get the sorted count of word and frequency
        Map<String, Integer> sorted = wordValueMap.entrySet().stream()

          .sorted(Collections.reverseOrder(Map.Entry.comparingByValue()))
                .collect(toMap(Map.Entry::getKey, Map.Entry::getValue, (e1,
        e2) -
                 > e2, LinkedHashMap::new));

        System.out.println();
        System.out.println("*Sorting Words Based on Count of First & Last
        Character*");
        System.out.println(sorted);
```

Sample Output:

```
{TECHNIQUE=9, THERE=9, SINGLE=8, TECHNOLOGY=8, IS=7, SIMPLICITY=7, MANAGEMENT=6,
EITHER=6, DEVELOPMENT=6, IN=5, PROMISES=5, NO=4, OR=4, ITSELF=4, BY=4, WHICH=2}
```

3. **Calculate the hash value to get frequency for each element.**

This function calculates hash value for each key based on length of key and g-Value. If there is any other key already stored at the position, then g-Value(firstletter) is incremented to 1 and the hashing is performed again.

```
public static int getFrequencyValue(String sortedKeys,
HashMap<Character, Integer> hm2, int dataLen,

String[] anArray, String[] strings) {

int length = sortedKeys.length();
int gFirst = hm2.get(sortedKeys.charAt(0));
int gSecond = hm2.get(sortedKeys.charAt(sortedKeys.length() - 1));

int frequencyValue = (length + gFirst + gSecond) % dataLen;

 if (anArray[frequencyValue] != null && gFirst <= 4) {

  hm2.put(sortedKeys.charAt(0), gFirst + 1);
 frequencyValue = getFrequencyValue(sortedKeys, hm2, dataLen,
                  anArray, strings);
       }

else if (anArray[frequencyValue] != null && gFirst > 4) {

 int prevIndex =(Arrays.asList(strings).indexOf(sortedKeys)) - 1;

try {
 String prevElement = strings[prevIndex];
 hm2.put(prevElement.charAt(0), gFirst + 1);

  frequencyValue = getFrequencyValue(prevElement, hm2,
                  dataLen, anArray, strings);

 if (anArray[frequencyValue] != null && gFirst <= 4) {

    hm2.put(prevElement.charAt(0), gFirst + 1);
   frequencyValue = getFrequencyValue(prevElement, hm2, dataLen,
                        anArray, strings);
 }

 else {
    anArray[frequencyValue] = prevElement;
 }

 }
```

```
catch (IndexOutOfBoundsException e) {
   System.out.println(e);

}

return frequencyValue;
}
```

Sample Output:

{OR=1, BY=2, IS=3, IN=4, NO=5, THERE=6, SINGLE=7, EITHER=8, ITSELF=9, TECHNIQUE=10, TECHNOLOGY=11, SIMPLICITY=12, MANAGEMENT=13, DEVELOPMENT=14, PROMISES=15, WHICH=16}

(The value represents the frequency of that respective element)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
OR	BY	IS	IN	NO	THERE	SINGLE	EITHER	ITSELF	TECHNIQUE	TECHNOLOGY	SIMPLICITY	MANAGEMENT	DEVELOPMENT	PROMISES	WHICH

The above table shows that minimal perfect hashing is achieved in allocating the elements to set of consecutive integers with collision and empty slots.

Code for Allocating Keys to Memory

Once the frequency of keys is achieved each and every element needs to be mapped to its respective memory in the index table based on frequency. Here Memory-A and Memory-B is created whose is created dynamically based on the size of data. In further when new element is coming in and there is no position for that to enter the memory is created dynamically. Also, a static index table is created using ‘HashMap’ to store the frequency and Memory Modules. High-frequency elements index to Memory-a and low-frequency elements index to Memory-b.

```
public static HashMap<Integer, Character> getMemoryModule() {
      HashMap<Integer, Character> indexTable = new HashMap<Integer,
                                                      Character>();

      for(int i=0;i<5;i++){
            indexTable.put(i, 'b');
      }
      for(int i=5;i<15;i++){
            indexTable.put(i, 'a');
      }
      return indexTable;
}
```

The above function creates the index such that less frequent keys stored in Memory-b and high frequency keys are stored in Memory-a. If any key with more frequency coming in then it is dynamically stored in the index table.

```
/*
* Iterate through set of keys in hashmap to index the f(x) value to get
Memory Module*/

    for (String key : hm.keySet()) {
       int value = hm.get(key);
       char memModule = 0;

 /** Check if the index table contains the frequency, if it presents
       * then the memory module is returned*/

            if (indexTable.containsKey(value)) {
                  memModule = indexTable.get(value);
            } else if (value > 4) {
                  memModule = 'a';
                  indexTable.put(value, 'a');
      }
```

Once the memory of element is achieved then hashing is performed to get the position of element to be stored in Memory. As the element is string, firstly it is converted to its ASCII value and then the hashing is performed.

```
/*
 * This function is used to get the hashCode for given key and
 frequency value*/

public static int toAscii(String key) {
      StringBuilder sb = new StringBuilder();
      String ascString = null;
      for (int i = 0; i < key.length(); i++) {
            sb.append((int) key.charAt(i));
      }
      ascString = sb.toString();

      return Math.abs((key == null ? 0 : key.hashCode()) ^ (ascString
     == null ? 0 : ascString.hashCode()));

 }
```

Finally, the position will be returned for respective key. If the position is empty, then the key will be stored in that location. If the position is not empty, then the value is hashed in Memory location ‘b’ to get the vacant position.

```
int position = MemoryMapping.toAscii(key);

if (memModule == 'a') {
    position = position % memoryA.length;
    if (memoryA[position] != null && memoryA[position] != key) {

          position = (MemoryMapping.toAscii(key)) % memoryB.length;
          memoryB[position] = key;
    }

   else {
          memoryA[position] = key;
    }
    }

 else if (memModule == 'b') {
    position = position % memoryB.length;

    if (memoryB[position] != null && memoryB[position] != key) {

          position = (MemoryMapping.toAscii(key)) % memoryB.length;
          memoryB[position] = key;
```

```
            }

            else {
                  memoryB[position] = key;
            }
            }
      }
```

Sample Output:

```
*Memory-A
Key[0]: null
Key[1]: TECHNIQUE
Key[2]: NO
Key[3]: null
Key[4]: DEVELOPMENT
Key[5]: SIMPLICITY
Key[6]: null
Key[7]: null
Key[8]: null
Key[9]: ITSELF
Key[10]: EITHER
Key[11]: null
Key[12]: SINGLE
Key[13]: THERE
Key[14]: null
Key[15]: WHICH

*Memory-B
Key[0]: null
Key[1]: PROMISES
Key[2]: null
Key[3]: null
Key[4]: null
Key[5]: null
Key[6]: BY
Key[7]: null
Key[8]: IN
Key[9]: null
Key[10]: TECHNOLOGY
Key[11]: null
Key[12]: null
Key[13]: IS
Key[14]: null
Key[15]: OR
```

Source Code Link:

https://github.com/cmoulika009/Thesis-Improving_Cuckoo_Hashing_with_Perfect_Hashing

CHAPTER 6

ANALYSIS

6.1 Comparative Analysis of Cuckoo Hashing and Our Approach

As discussed earlier our thesis improves the efficiency of memory utilization and keys hit ratio by combining Cuckoo Hashing and Perfect Hashing. As part of this section, we perform analysis how exactly our approach works. Cuckoo Hashing works efficient for small amount of data, i.e. in Cuckoo Hashing for each key two locations are maintained for which two hash functions are required. When an element ‘x’ is hashed and gets position 10 which is already occupied by element ‘y’ then hashing is performed on element ‘y’ to gets its alternate location. Now ‘y’ will be moved to alternate location and ‘x’ is stored in actual location. We should repeat this process (bouncing between tables), until all elements stabilize. Below is the java code that is used to get alternate locations for given key in the table.

Code:

```
//Moving old keys to alternate locations
 public final boolean equals(Object o) {
       if (!(o instanceof Map.Entry))
             return false;
       Map.Entry e = (Map.Entry) o;
       Object k1 = getKey();
       Object k2 = e.getKey();
       if (k1 == k2 || (k1 != null && k1.equals(k2))) {
             Object v1 = getValue();
             Object v2 = e.getValue();
             if (v1 == v2 || (v1 != null && v1.equals(v2)))
                   return true;
       }
       return false;
       }
```

But in this approach, the insertions to alternate locations may run into cycles. If that happens then we need to rehash by choosing a new h_1 and h_2, and we need to insert all the elements back into the table. Multiple rehashes are required to make it success. So here there might be possibility to revisit the same slot twice for a successful insertion. Cycles only arise if we revisit the same slot with the same element to insert. So Cuckoo Hashing will enter infinite loops which inserting keys into memories which leads to collision.

In our approach as we used Cichelli's method to achieve Minimal Perfect Hashing, there is no possibility of collision while allocating keys to 0 to (n-1) distinct integers without any collisions. The hash method that is used in this algorithm takes care of all collisions that will occur in allocating frequencies to distinct integers.

H(word) = (length(word) + g(firstletter(word)) + g(secondletter(word)) % TSize

So, once the frequencies are calculated using Perfect Hash Function, the frequency is indexed to memory. High-frequency elements are stored in Memory-a and low-frequency elements are stored in Memory-b. Once the entire implementation is completed successfully, the key lookup is done in constant access time and memory utilization is efficient using this approach. Keys hit ratio is also high as keys are allocated based are their frequencies using Minimal Perfect Hashing.

CHAPTER 6

CONCLUSION

In this thesis, we studied about Cuckoo Hashing and Perfect Hashing. We studied different implementations of Cuckoo Hashing and how Parallel-d-Pipeline is one of the updated implementation of Cuckoo Hashing. We also studied Perfect Hashing and its variations such as Minimal Perfect Hashing. We discussed about various disadvantages of Cuckoo Hashing implementation, which results in cycles during memory allocation. Further, we discussed how we used perfect hashing mechanism to improve Cuckoo Hashing by considering 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.

REFERENCES

[1] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," 29th Annual Symposium on Foundations of Computer Science, White Plains, NY, vol. 23, no. 4, pp. 524-531, 1988.

[2] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-Based Techniques for High-Speed Packet Processing", In: Cormode G., Thottan M. (eds) Algorithms for Next Generation Networks. Computer Communications and Networks. Springer, London, pp 181-218,2010.

[3] D. Belazzougui, F.C. Botelho, and M. Dietzfelbinger, "Hash, Displace, and Compress", In: Fiat A., Sanders P. (eds) Algorithms - ESA 2009. ESA 2009. Lecture Notes in Computer Science, vol. 5757. Springer, Berlin, Heidelberg, 2009.

[4] G. Levy, S. Pontarelli and P. Reviriego, "Flexible Packet Matching with Single Double Cuckoo Hash," in *IEEE Communications Magazine*, vol. 55, no. 6, pp. 212-217, doi: 10.1109 /MCOM.2017.1700132, 2017.

[5] P. Gupta and N. McKeown, "Algorithms for Packet Classification," in *IEEE Network*, vol. 15, no. 2, pp. 24-32, doi: 10.1109/65.912717, Mar/Apr 2001.

[6] R. Pagh, and F.F. Rodler. "Cuckoo Hashing". Algorithms — ESA 2001. ESA 2001. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, vol. 2161, pp. 121–133. ISBN 978-3-540-42493-2,2001.

[7] R. Raman, "The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting", Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, Berlin, Heidelberg, vol. 472, pp. 9-11,1990.

[8] R. Raman, "Priority Queues: Small, Monotone and Trans-dichotomous", Proc. 1996 European Symp. on Algorithms, Lecture Notes in Computer Science, vol. 1136, pp. 121-137,1996.

[9] S. Pontarelli, P. Reviriego and J. A. Maestro, "Parallel d-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput," in *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 326-331, Jan. 1 2016.

[10] F. P. J. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," in *Computer*, vol. 20, no. 4, pp. 10-19, April 1987.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Introduction to Algorithms" (3rd

ed.). Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384,2009.

[12] M. Mitzenmacher, "Some Open Questions Related to Cuckoo Hashing", In: Fiat A., Sanders P. (eds) Algorithms - ESA 2009. ESA 2009. Lecture Notes in Computer Science, vol 5757, Springer, Berlin, Heidelberg, 2009.

[13] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time", Theory of Computing Systems, vol. 38, no. 2, pp. 229-248, 2005.

[14] B. Jenkins, "Order-preserving Minimal Perfect Hashing", in Dictionary of Algorithms and Data Structures [online], V. Pieterse and P.E. Black, eds. 14 April 2009. Available from: https://www.nist.gov/dads/HTML/orderPreservMinPerfectHash.html

[15] E.A. Fox, Q. Chen, and A. M. Daoud, "Order-preserving Minimal Perfect Hash Functions and Information Retrieval", ACM Transactions on Information Systems, New York, NY, USA: ACM, pp. 281–308,1991.

[16] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator", C++ Report, SIGS, Vol. 10, No. 10, December 1998. Available from: http://www.cs.wustl.edu/~schmidt/PDF/gperf.pdf

VITA

Moulika Chadalavada was born on August 25, 1992, in Andhra Pradesh, India. She completed her Bachelor's degree in Information and Communication Technology from SASTRA University in Thanjavur, Tamilnadu.

After completing her under-graduation, she worked as Software Developer in Tata Consultancy Services, Hyderabad for 3 years. To enhance her career further she started her masters in Computer Science at the University of Missouri-Kansas City (UMKC) in Fall 2016, specializing in Software Engineering. Upon completion of her requirements for the Master's Program, Moulika Chadalavada plans to work as a Software Developer.