Performance Analysis of Sequential Cuckoo Hashing under Light of Switching Cost

Rajeev Ranjan Kumar Tripathi, Pradeep Kumar Singh, Sarvpal Singh

Abstract

Cuckoo Hashing has been considered the state of the art since its origin. In Cuckoo Hashing, the search operation is executed sequentially in the tables, and thus, Cuckoo Hashing is Sequential Cuckoo Hashing. Based on the size of the table, Cuckoo Hashing is implemented in two ways: Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing. Cuckoo Hashing suffers from higher insertion latency, inefficient memory usage, and high data migration. This work investigates the performance of both versions of Cuckoo Hashing under the light of two newly proposed performance indicators: Degree of Dexterity and Table Reference Count per key.

Keywords: Symmetric Sequential Cuckoo Hashing, Asymmetric Sequential Cuckoo Hashing, Degree of Dexterity, Table Reference Count per Key.

MSC 2020: 68P05, 68P10, 68P20.

ACM CCS 2020: Information systems—Database management-system engines, Information systems—Information Storage Systems.

1 Introduction

The continuous and rapid growth in Mobile Computing, the Internet of Things (IoT), Big Data, Cloud Computing, Distributed Systems, and Social Media has attracted the public and industries because there is a significant amount of data flow. Data generation is performed on a wider scale. Expeditious improvements have been made in data

doi:10.56415/csjm.v34.06

storage techniques, and data centers are examples of this success. Data insertion, deletion, update, and search are the common operations that are performed on stored data. The classical problem in Computer Science is to perform retrieval operation in $O(1)$. This kind of search is frequently used in data-based industries, data dictionaries, and symbol tables maintained by compilers. New application areas like Big Data, Health Informatics, and Content-Based Image Retrieval have highlighted searching as a major challenge [1]–[3].

Information is stored in the form of records in computer memory. Each record has a unique key that identifies the record uniquely, and this key is known as the primary key. The primary key may be derived from information available in the records, or the administrator may explicitly assign it. The size of the primary key does not play any role. The searching algorithm has to receive the key K, as input and return the record that has the key K. Hashing is an improved searching technique where data is placed at the address (sometimes also called index/hash code or hash) that is obtained by a hash function as $h = H(K)$, where H is the hash function, h is the hash code for the key (K). The h and K can be observed as an ordered pair of (hash, key), *i.e.*, (h, K). Searching using hashing requires $O(1)$ time ideally. The hash function (H) may assign the same hash codes to distinct keys K_1 and K_2 as (h, K_1)=(h, K_2). Thus, the keys K_1 and K_2 are often called synonyms [4] and the assignment of the same hash code to distinct keys is called **data collision**.

While surveying on Cuckoo Hashing, various existing works have been encountered like *d-ary Cuckoo Hashing* [5], *Cuckoo Hashing with Stash* [6], *Lock-Free Bucketized Cuckoo Hashing* [7], *Lock Free Cuckoo Hashing* [8], *Backyard Cuckoo Hashing* [9], *Parallel Cuckoo Hashing* [10], *Generalized Cuckoo Hashing* [11], and *Cache-Conscious Cuckoo Hashing* [12]. From the literature survey, the authors are motivated to perform work to clarify the misconceptions about Cuckoo Hashing. **The most common misconception is the performance of Cuckoo Hashing in terms of searching time. The $O(1)$ performance in searching creates doubt that both tables used in Cuckoo Hashing are being accessed in parallel.** Cuckoo Hashing, in its purest form, always uses sequential access (Cuckoo Hashing

is an acronym of Sequential Cuckoo Hashing) [13]. Cuckoo Hashing consumes 2 probes in the worst case to declare whether the search is successful or not, and that's why its performance is of order $O(1)$. This analysis does not consider the switching cost consumed by the Cuckoo Hashing while searching keys from a large dataset in a one-to-one fashion. The switching cost degrades the overall performance of searching. This case is explained by a story of **Two Shops Example** in the next section. Based on the size of hash tables, Cuckoo Hashing may be either symmetric or asymmetric. This work is dedicated to comparing the performance of Symmetric Sequential Cuckoo Hashing, Asymmetric Sequential Cuckoo Hashing, Random Probing, and Quadratic Probing.

The major contributions of this work are as follows:

1. In this paper, the performance analysis of Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing is investigated, and their prototypes are implemented under a single thread. Major challenges of Cuckoo hashing are *inefficient memory usage, higher insertion latency*, and *data migration*. In addition, a new challenge, **high switching cost**, has been identified in both versions of Sequential Cuckoo Hashing.

2. Authors introduce two new performance indicators: 1) *Degree of Dexterity*(η) and 2) *Table Reference Count per Key*, denoted as $\frac{\mathbf{T.R.C}}{\mathbf{Key}}$. The *Degree of Dexterity* is the reciprocal of the sum of average searching time and insertion latency involved in the hashing process, while *Table Reference Count per key* represents "how many times tables have to be consulted to search a key?". *Table Reference Count per key is a one-of-its-kind and novel way of analyzing hashing techniques that use more than one hash tables.*

3. To reduce insertion latency, data migration cost, and insertion failure avoidance, the concept of **stash memory** was proposed by several researchers [14]–[16]. In this work, the authors notice that data loss observed in Cuckoo Hashing decreases as "MaxLoop" increases. The experimental result endorses the

claim made by the researchers in the context of "stash memory". Cuckoo Hashing uses two hash functions from the Family of Universal Hash Functions. *Universal hash functions* by property, always report data loss of only one key. Thus, *during the experiment, authors realize that by maintaining a high value of "MaxLoop", execution of the costly rehash operation may be avoided in both versions of Cuckoo Hashing.* The authors are the first to address this issue in the context of Cuckoo Hashing to the best of their knowledge.

2 Literature Survey on Hashing Technique

The most common practice is to use the hashing technique with some collision resolution techniques. Some of the existing literature is discussed in Table 1 including their methodology and shortcomings. This section also discusses the Cuckoo Hashing in detail.

Cuckoo Hashing is a nature-inspired hashing technique based on randomization [17]–[19]. Cuckoo Hashing uses two tables: denoted as T_1 and T_2 of same or different sizes and two hash functions: H_1 and H_2. If the size of the tables is the same, it is called Symmetric Cuckoo Hashing; otherwise, it is referred to as Asymmetric Cuckoo Hashing. The hash functions distribute the universe of keys $\mathbf{U}$ over T_1 and T_2, respectively. A key K is stored either at $T_1[H_1(K)]$ or $T_2[H_2(K)]$. Thus, retrieval of K requires two lookup operations in both tables under the worst case. Lookup operation may be performed either in parallel or sequential manners and, based on this, Cuckoo Hashing is categorized as **Parallel Cuckoo Hashing** and **Sequential Cuckoo Hashing** (further categorized as Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing, based on the size of tables). Inventors of Cuckoo Hashing suggested a relation between the size of tables (n) and the number of keys ($\mathcal{R}$) as $n \geq (1+\epsilon)\,\mathcal{R}$, where ϵ is a constant and $\epsilon \geq 0$ [20]–[22] in case of Symmetric Sequential Cuckoo Hashing. In the Asymmetric version of Cuckoo Hashing, the size of a table is kept twice the size of another table. Thus, the memory requirement in Asymmetric Sequential Cuckoo Hashing is more than in Symmetric Sequential Cuckoo Hashing. When a collision is ob-

served, to make room for freshly arriving keys, Cuckoo Hashing kicks away existing keys from their places. *The endless kicking-out operation can be avoided by setting a threshold on "MaxLoop".* When this "kicking-out" operation reaches the "MaxLoop", a rehash operation is performed. If T_1 and T_2 are almost half full and hash functions, H_1 and H_2, are chosen from Universal Family of Hash Functions of order $(O(1), O(log_n))$, then there is a probability of $O(\frac{1}{n})$ that a key (K) is not placed in both, T_1 and T_2 [13], [23], [24]. Further, the inherent challenges associated with Cuckoo Hashing are a) **inefficient memory usage**, b) **data migration**, and c) **higher insertion latency** [25]–[29].

In K-ary Cuckoo Hashing ($K > 2$), K hash functions generate K indices to perform lookup operations in K tables. This approach offers worst-case lookup time and space complexity. Instead of using multiple tables, a single table (of size K) can be logically partitioned into $\frac{K}{L}$ logical tables, and each logical table has the size of L blocks. This modification avoids the additional costs involved in accessing the tables that report unsuccessful lookup operations and includes only the cost involved in key comparisons [30]. In case of collision, data migration takes place, and it is performed up to the set value of "MaxLoop". Thus, the higher value of "MaxLoop" leads to high data migration cost. Additionally, the way in which data collision is dealt with the Cuckoo Hashing demands high insertion latency [25], [31]. All steps for lookup and insertion operations are given in Algorithms 1 and 2.

Algorithm 1: Lookup Function

Input: key:K;
Output: If K may be found either in T_1 or T_2, on success it returns K;

```
1 if K = T1[H1(K)] then
2     return K;
3 if K = T2[H2(K)] then
4     return K;
5 end
```

Algorithm 2: Insert(K)

```
1  if lookup(K) then
2      return
3  loop MaxLoop times
4      if T1[H1(K)] =⊥
5          T1[H1(K)] ← K;
6          return;
7      else
8          K ↔ T1[H1(K)];
9      if T2[H2(K)] =⊥
10         T2[H2(K)] ← K;
11         return;
12     else
13         K ↔ T2[H2(K)];
14 end loop
15 rehash(); insert(K);
16 end
```

In Sequential Cuckoo Hashing, the lookup operation can be initiated from any table, and the second table is only consulted when the lookup operation is declared unsuccessful in the first table (it causes two searching times for the same set of keys). Insertion operations in both Parallel Cuckoo Hashing and Sequential Cuckoo Hashing are the same [32]–[34].

Table 1. Summary of previous works on Hashing Technique with their drawbacks

Existing Method	**Approach**	**Drawback**
Division Method [35]–[37]	Based on modulo m division (m may be either a prime number or not)	Poor performance on randomly chosen table size, Increased collision probability, Potential for poor performance with certain key distributions
Mid-Square Method [37], [38]	Based on squaring the key and extracting the middle digits	Limited distribution, Biased distribution, Vulnerability to poor choice of key, Sensitivity to table size, Limited applicability, Difficult to handle collisions
Folding Method [37], [38]	Based on key segmentation	Uneven distribution of keys, Sensitivity to key length, Complex in implementation, Limited applicability, Difficult to handle collisions, Potentially biased
Linear Probing [39], [40]	In the presence of collision, keys are stored at some other locations. Empty slots are searched linearly.	Clustering issues, Performance degradation with an increase in load factor, Poor worst-case, Performance of order O(n), Achieves low load factor in a dynamic environment
Quadratic Probing [41]	In the presence of collision, keys are stored at some other locations. Empty slots are searched in a quadratic fashion.	Suffers from secondary clustering, Dense proximity of collisions (because of uneven key distribution)
Random Probing [42]	In the presence of collision, keys are stored at some other locations. Empty slots are searched randomly.	Poor worst-case performance, Probe sequences may visit the same set of slots

Continued on next page

Table 1 – continued from previous page

Existing Method	Approach	Drawback
Double Hashing [43]	Uses another hash function to deal with data collision	Computationally complex, Challenges in selection of second hash function, Prone to clustering, Sensitivity to load factor, Increased memory overhead
Chaining [44]–[46]	Based on linked list. Keys are stored in a linked list	Increased memory usage, Poor cache performance, Difficulty in balancing load factor, Extra pointer overhead, Difficulty in parallelization, Poor worst-case performance, Difficulty in deletion of keys
Coalesced Hashing [47]	Coalesced hashing is an efficient version of Chaining	Dynamic resizing of the hash table is difficult, Potential for fragmentation, Increased memory overhead, Limited applicability, Poor worst-case performance
Cuckoo Hashing [17]–[19], [25]–[29]	Based on Universal Hash Functions and randomization. Conventionally uses two tables.	The size of tables may be either equal or unequal with high insertion latency, Inefficient memory usage, High data migration cost

3 Work Done: Performance Analysis of Sequential Cuckoo Hashing under Light of Switching Cost

The work is dedicated to estimating the switching cost which is involved in Sequential Cuckoo Hashing. Section 3.1 raises the issue of "switching cost" with an example of **Two Shops Example**. Section 3.2, sheds light on the mathematical analysis to estimate the switching cost and its implication on searching time. Section 3.3 predicts the data collision that occurred in Sequential Cuckoo Hashing using *bin and ball model*.

Section 3.4 deals with the hit ratio and miss ratio in tables used by Sequential Cuckoo Hashing. Section 3.5 deals with space complexity, and Section 3.6 deals with time complexity.

3.1 Two Shops Example

In a colony, there are two grocery shops run by *Alice* and *Bob*. *Adam* is a customer of both shops and he finds **80**% of his needs at *Alice's* shop. The remaining **20**% of goods items are only available at *Bob's* shop. Adam knows these statistics well, and he always visits *Alice's* shop first. *Sophia* and *Samuel* are twins in the family of *Adam*. One day, Adam asks *Sophia* and *Samuel* to purchase some grocery items with the instruction that *Sofia* will always start purchasing from *Alice's* shop while *Samuel* from *Bob's* shop, and at a time, only one item has to be purchased. If an item is not found in the shop, then they have to visit another shop. *Adam* provides **10** item-names to both *Sophia* and *Samuel* one by one and starts recording the time consumed in each transaction of *Sophia* and *Samuel*. Undoubtedly, *Sophia* consumes less time than *Samuel* in overall transactions. This is because *Sophia* has visited *Alice's* shop **10** times, and only **2** times she has visited *Bob's* shop. Thus, *Sophia* visits both shops for each item only **1.2** times, while *Samuel* consumes **1.8** shop visit per item on average.

3.2 Estimation of Switching Cost in Sequential Cuckoo Hashing

In Parallel Cuckoo Hashing, the lookup operation is performed simultaneously in both tables. As Cuckoo Hashing stores keys according to the computed hash codes, a single probe is required in searching for a key. Thus, the time requirement of searching a key in Parallel Cuckoo Hashing is $\mathbf{O(max(t_1, t_2))} = \mathbf{O(1)}$, where t_1 and t_2 are the searching time consumed in the tables T_1 and T_2, respectively. The conceptual view of Parallel Cuckoo Hashing is shown in Figure 1.

In the case of Sequential Cuckoo Hashing, lookup operations in the tables are not performed in parallel but in a sequential manner. Thus, the time requirement in searching for a key in Sequential Cuckoo Hashing cannot be $\mathbf{O(1)}$. Let P_1 and P_2 be the probabilities of finding a

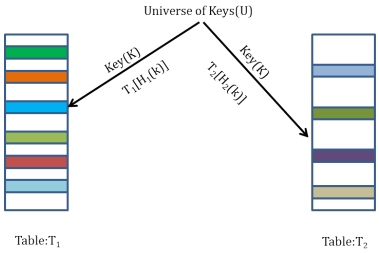

Figure 1. Conceptual View of Parallel Cuckoo Hashing

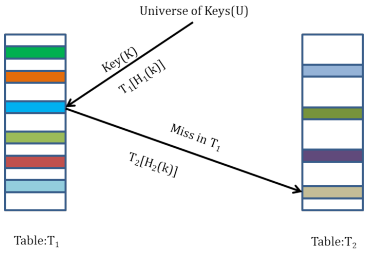

Figure 2. Conceptual View of Sequential Cuckoo Hashing

key in the tables T_1 and T_2, respectively. A key can be placed only on a table. This analysis assumes that Sequential Cuckoo Hashing does not report any data loss, and thus, $P_1 = (1-P_2)$ (hit ratio of $T_1 = (1-$ miss ratio of $T_2)$). Sequential Cuckoo Hashing assigns the computed hash codes to the keys, and thus, the lookup operation demands only one probe to ensure the availability of the key in a table. Let t be the cost of this single probe, then the estimated cost of search $T_{\phi 1}$ can be computed as $P_1 \times t + P_2 \times (t+t) = P_1 \times t + 2 \times (1-P_1) \times t = t \times (2-P_1)$ if lookup operation is initiated from the table T_1. If table T_2 is investigated first, then $T_{\phi 2} = t \times (2 - P_2)$. As $P_1 \neq P_2$, thus, $T_{\phi 1} \neq T_{\phi 2}$, and searching depends on the order of lookup in Sequential Cuckoo hashing.

The conceptual view of Sequential Cuckoo hashing is shown in Figure 2. Table T_2 is consulted when a *miss* is reported during a search of a key K in the table T_1. Transferring of control from T_1 to T_2 involves additional overhead in terms of **switching cost**. If the universe of keys $(\mathcal{U})$ is $|\mathcal{U}| = n$ and the lookup operation is initiated to find all the keys of $\mathcal{U}$, in this case, the switching cost cannot be ignored while estimating the total searching cost. To estimate the involved switching cost, this work proposes **T.R.C.** (Total Reference Count), a new performance indicator. Let the search start from Table T_1, and for every key, T_1 is searched first. If this search is unsuccessful, T_2 is searched then. Thus, we can count the total number of table accesses during the search, and it is $TRC_{T1} = n + n \times (1-P_1)$, where TRC_{T1} is the total reference count when T_1 is consulted first. If T_2 is consulted first, the total reference count is $TRC_{T2} = n + n \times (1 - P_2)$. Now, we can compute the average

TRC, which is the ratio of the total reference count to $|\mathcal{U}| = n$. Now, in the case of Sequential Cuckoo Hashing, there are two average TRC, $TRC_{AvgT1} = \frac{n+n\times(1-P_1)}{n}$ and $TRC_{AvgT2} = \frac{n+n\times(1-P_2)}{n}$ when search is started from T_1 and T_2, respectively. If α is the cost of a single switch, then searching costs of the keys, T_{T1} and T_{T2} are $T_{\phi 1} + TRC_{AvgT1} \times \alpha$ and $T_{\phi 2} + TRC_{AvgT2} \times \alpha$, respectively, when searching is initiated from T_1 and T_2. Thus, T_{T1} and T_{T2} are $(2-P_1)\times(t+\alpha)$ and $(2-P_2)\times(t+\alpha)$ for a key. Based on the above discussion, we can propose Lemma 1.

Theorem 1. *Sequential Cuckoo Hashing offers two searching times for the same set of* $\mathcal{U}$.

Proof. Let T_{T1} and T_{T2} be the searching cost when search operations are initiated from the tables, T_1 and T_2, respectively. If probabilities of finding a key in T_1 and T_2 are P_1 and P_2, then T_{T1} and T_{T2} are $(2 - P_1) \times (t + \alpha)$ and $(2 - P_2) \times (t + \alpha)$, respectively. Here, t is the cost of a single probe consumed searching for a table key, and α is the switching cost. Thus, "Sequential Cuckoo Hashing offers two searching times for the same set of $\mathcal{U}$". The experimental proof is shown in Table 3. **Theorem 1 endorses the claim of [32]**. □

Lemma 1. *Sequential Cuckoo Hashing offers two* ***switching costs*** *for the same* $\mathcal{U}$.

Proof. Switching cost is estimated with the help of the average Table Reference Count. The $TRC_{AvgT1} = \frac{n+n\times(1-P_1)}{n}$ and $TRC_{AvgT2} = \frac{n+n\times(1-P_2)}{n}$ when search is started from T_1 and T_2, respectively as stated above. Thus, "Sequential Cuckoo Hashing offers two **switching costs** for the same set of $\mathcal{U}$." The experimental proof is shown in Table 3. □

3.3 Prediction of Collision

Bin and Ball model can be used to estimate an average number of collisions [48]. Let there be $\mathcal{X}$ balls and $\mathcal{Y}$ bins with $\mathcal{Y} > \mathcal{X}$. The $\mathcal{X}$ balls are randomly thrown into the $\mathcal{Y}$ bins. The first ball is placed in a bin; the remaining ($\mathcal{X}$-1) balls have an equal probability of falling into the same bin, which is $\frac{1}{\mathcal{X}}$. Thus, the average number of collisions with

the first ball will be $\frac{(\mathcal{X}-1)}{\mathcal{Y}}$. The second ball falls into the same bin, and the remaining ($\mathcal{X}$-2) balls have an equal probability of falling into the same bin, which is $\frac{1}{\mathcal{X}}$. Thus, average number of collisions, denoted as Avg_{cols}, with the second ball will be $\frac{(\mathcal{X}-2)}{\mathcal{Y}}$.

$$Avg_{cols} = \sum_{i=1}^{(\mathcal{X}-1)} \frac{i}{\mathcal{Y}} = \frac{\mathcal{X}(\mathcal{X}-1)}{2\mathcal{Y}}. \tag{1}$$

The number of collisions given in Eq. 1 is used to perform the quantitative analysis.

3.4 Hit Ratio and Miss Ratio

In Cuckoo Hashing, if the search operation in a table fails, another table is consulted. The probability of finding a key in a table is $\frac{1}{2}$ in Cuckoo Hashing [49]. *Hit ratio* and *miss ratio* both affect the searching performance of Sequential Cuckoo Hashing. *Hit ratio* is defined as the ratio of *"total number of keys found in a table to the number of times a table is consulted"*. Miss ratio is defined as *"total number of keys not found in the table to the total number of times a table is consulted"*. A simplified version of Cuckoo Hashing has been proposed that maintains a single table instead of two tables, so that the probability of finding an empty location improves. This modification significantly improves the performance of insertion and searching cost [50]. In Sequential Cuckoo hashing, **miss ratio ≠ (1-hit ratio)**. This is because data loss is observed in Sequential Symmetric Cuckoo Hashing. In this work, **Table Reference Count (T.R.C.)** is used to calculate the searching cost, and **T.R.C.** is the sum of all *hits* and *misses* of the tables used in a scheme. $\frac{T.R.C.}{key}$ is the ratio of *T.R.C. to the total number of keys.* $\frac{\mathbf{T.R.C.}}{\mathbf{key}}$ shows that *"how many reference counts of tables are required to search a key on an average basis?"* When hashing scheme uses more than one hash table, $\frac{T.R.C.}{key}$ cannot be ignored while calculating the searching time.

3.5 Space Complexity

In Symmetric Sequential Cuckoo Hashing, the size of a table is maintained as $\mathcal{S}$, and this is considered a benchmark for the rest of the techniques, such as Random Probing, Quadratic Probing, and Asymmetric Sequential Cuckoo Hashing. Thus, the space complexity of Symmetric Sequential Cuckoo Hashing is $O(2\mathcal{S})$. Random Probing and Quadratic Probing use only one hash table; thus, the space complexity of Random Probing and Quadratic Probing is $O(\mathcal{S})$. In Asymmetric Sequential Cuckoo Hashing, the size of one table is kept as $\mathcal{S}$, while the size of another table is $C\mathcal{S}$, where C is a constant with value 1.1. Thus, overall space complexity of Asymmetric Cuckoo Hashing is $O(C_1\mathcal{S})$, where C_1 is a constant having value 2.1.

The space complexity of Asymmetric Sequential Cuckoo Hashing and Symmetric Sequential Cuckoo Hashing is nearly equal. Though allocation of large-sized-contiguous memory is addressed as a major problem in the past, it is not currently considered as a major problem [50]–[53]. The load factor (λ) is a performance indicator in the hashing technique that measures memory utilization. The λ is the ratio of the total number of keys, n, to the total number of memory locations, m, used in the hashing scheme, and thus, $\lambda = \frac{n}{m}$. Theoretically, $\lambda = \frac{1}{2}$ in the case of Symmetric Sequential Cuckoo Hashing.

Lemma 2. *Memory utilization in both Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing is always less than* 50*%.*

Proof. Space complexity of Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing is $O(2S)$ and $O(C_1S)$, respectively, as discussed in the Section 3.5. Let n be the total number of keys to be stored in both Symmetric Cuckoo Hashing and Asymmetric Cuckoo Hashing. As $S >> n$, the load factor (λ) is always less than 0.5, and thus, the memory utilization in both Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing is always less than 50%.

□

3.6 Time Complexity

Quadratic Probing and Random Probing do not offer the performance of $O(1)$, while Cuckoo Hashing offers the performance of $O(1)$ even in the worst case. The performance of Cuckoo Hashing is better than the performance of Quadratic Probing and Random Probing because it exhibits *high associativity* between keys and computed hash codes.

The searching time consumed in the Parallel Cuckoo Hashing is always less than the searching time consumed in the Sequential Cuckoo Hashing. In Sequential Cuckoo Hashing, the sequence(S_i) (defined as $T_i \rightarrow T_j$, where $j = 2 \iff i = 1$ and $j = 1 \iff i = 2$) determines the searching cost as discussed previously in Theorem 1.

Corollary 1.1. *Sequential Cuckoo Hashing performs searching in* $O(1)$ *iff a predefined sequence is followed.*

Proof. Let the sequence (S_1) be defined as

$$S_1 : T_1 \rightarrow T_2. \tag{2}$$

The $\rightarrow$ represents the "Search Before" relationship, and $T_1 \rightarrow T_2$ tells that the search operation first takes place in the table T_1 and then in the other table T_2. P_1 and P_2 are the probabilities of finding a key in T_1 and T_2, respectively. For α denoting the cost of the single switch from T_1 to T_2, the total searching cost (T_{T1}) is $(2 - P_1) \times (t + \alpha)$, where t is the cost of single probe consumed in searching a key in a table. Similarly, for the sequence (S_2),$S_2 : T_2 \rightarrow T_1$, the total searching cost(T_{T2}) is $(2 - P_2) \times (t + \alpha)$. Thus, searching time in Sequential Cuckoo Hashing is $O(C \times (t + \alpha))$, where C is a constant having value $(2-P_i)$ according to the sequence S_i. If the lookup operation is initiated from a table that holds a larger number of keys, the time consumed in Sequential Cuckoo Hashing is of order $O(1)$. Conclusively, we can assert "*Sequential Cuckoo Hashing performs searching in a* $O(1)$ *iff a predefined sequence is followed*". The experimental proof is shown in Table 3. □

4 Experimental Results and Discussion

This section presents a quantitative analysis of the performance of Symmetric Sequential Cuckoo Hashing, Asymmetric Sequential Cuckoo Hashing, Random Probing, and Quadratic Probing.

4.1 Experimental Setup and Data Structure Used

During the evaluation, this work considered a pure random dataset from Kaggle (10^6 in numbers, for every scheme) [54] that possesses a key length of 15 digits. Prototypes of Symmetric Sequential Cuckoo Hashing, Asymmetric Sequential Cuckoo Hashing, Random Probing, and Quadratic Probing are implemented under GCC Compiler, 11.3.0 on $i7$ processor of 3.40 GHz with 8 GB DRAM at Ubuntu Linux 22.04.1. Additionally, *clock()* of *time.h* is used to record the consumed time. During the experiment, two arrays of the same size (1000033) are implemented for Symmetric Sequential Cuckoo Hashing. It is to be noted that, Paugh and Rodler [55] recommended maintaining the size of a table twice the size of another table for Asymmetric Sequential Cuckoo Hashing. In the case of Asymmetric Sequential Cuckoo Hashing, the authors maintain two arrays of size 1000033 and 1099997, respectively. By maintaining two tables with a difference of $99,964$ in size, the authors are interested in observing the impact of table size on the *Data Loss* in the case of Asymmetric Sequential Cuckoo Hashing. Two hash functions ($(a * key + b)\%M$ and $(key/M)\%M$), where M is a prime number and a and b are constants) from the Family of Universal Hash Functions are used to compute the hash code.

4.2 Quadratic Probing and Random Probing

Performance analysis of Random Probing and Quadratic Probing is presented in Table 2. Data Loss, Collision Rate, Load Factor, Average Searching Time, Insertion Latency, and Degree of Dexterity(η) are the performance indicators. If *Data Loss* is ignored, Quadratic Probing is faster than Random Probing. Random Probing better utilizes the memory locations of the assigned hash table than Quadratic Probing.

Table 2. Performance Analysis of Random Probing and Quadratic Probing

Parameter	Scheme	
	Random Probing	Quadratic Probing
Data Loss	35195	367520
Collision Rate	0.460054	0.301589
Load Factor	0.92011	0.60318
Average Searching Time (in seconds)	0.061441	0.031863
Insertion Latency (in seconds)	0.028302	0.041207
Degree of Dexterity(η)	11.14289	13.68557

4.3 Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing

In this section, performance evaluation of Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing is done. While evaluating the performance of both versions of Cuckoo Hashing, the authors use different performance indicators, including a) *Collision Rate*, b) *Hit Count and Miss Count*, c) *Memory Utilization*, d) *Data Loss*, e) *Average Searching Time*, f) *Insertion Latency*, and g)*Degree of Dexterity (η)*. During the trials, a wide range of MaxLoop (ML) values was considered. For Symmetric Sequential Cuckoo Hashing, ML ranged from $\mathbf{ML = 10}$ to $\mathbf{ML = 220}$ with an interval of **10**. For Asymmetric Sequential Cuckoo Hashing, ML ranged from $\mathbf{ML = 10}$ to $\mathbf{ML = 95}$ with an interval of **5**. At $\mathbf{ML = 200}$ and $\mathbf{ML = 95}$, Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing show data loss of 1 and 0 key, respectively. No consistent effect of ML is observed on *Insertion Latency, Searching Time, and Degree of Dexterity* in the case of both Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing. The ML influences only *Data Loss and Memory Utilization*. The ML increases, *Data Loss* decreases (up to a certain value of ML only); thus, *Memory Utilization* increases. Table 3 shows the performance of Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing on the various performance indicators. The claim made by [49] is not observed in the case of Symmetric Cuckoo Hashing. The current work emphasizes the avoidance of execution of costly rehash operations by setting a high value of "MaxLoop" in the case of Sequential Cuckoo Hashing.

The authors are the first to address this issue in the context of Cuckoo Hashing to the best of their knowledge.

4.4 Comparison of Quadratic Probing, Random Probing, Symmetric Cuckoo Hashing, and Asymmetric Sequential Cuckoo Hashing

In terms of *Average Searching Time* and *Data Loss*, Asymmetric Sequential Cuckoo Hashing is better than the rest of the three schemes: Symmetric Sequential Cuckoo Hashing, Random Probing, and Quadratic Probing. Memory utilization of Random Probing is the best among all schemes. In terms of *Insertion Latency*, the performance of Random Probing is better than the rest of the techniques, as shown in Table 4. Asymmetric Sequential Cuckoo Hashing supersedes Symmetric Sequential Cuckoo Hashing in terms of η; however, the Quadratic Probing shows a high value of η among all schemes.

Table 3. Performance Analysis of Symmetric Sequential Cuckoo Hashing and Asymmetric Sequential Cuckoo Hashing

Parameter	**Scheme**			
	Asymmetric Sequential Cuckoo Hashing		Symmetric Sequential Cuckoo Hashing	
	Table1	Table2	Table1	Table2
Hit Count	343803	656197	367773	632226
Miss Count	656197	343803	632227	367773
Collision Rate	0.328087173	0.156274063	0.32	0.18
T.R.C./Key	1.656197	1.343803	1.632227	1.367773
Average Searching Time (in seconds)	0.05839825	0.045416739	0.057741409	0.0465724
Insertion Latency (in seconds)	0.074727	0.077076	0.077688	0.07855
Degree of Dexterity	7.511722983	8.163749168	7.383920558	7.992174063

Table 4. Performance Analysis of Symmetric Sequential Cuckoo Hashing, Asymmetric Sequential Cuckoo Hashing, Random Probing and Quadratic Probing

Parameter	**Scheme**					
	Asymmetric Cuckoo Hashing		Symmetric Cuckoo Hashing		Probing Technique	
	Table1	Table2	Table1	Table2	Random Probing	Quadratic Probing
Average Searching Time (in Seconds)	0.05839825	0.045416739	0.057741	0.0465724	0.0614413	0.03186265
Insertion Latency (in Seconds)	0.074727	0.077076	0.077688	0.07855	0.028302	0.041207
Degree of Dexterity (η)	7.511722983	8.163749168	7.383921	7.992174063	11.14289312	13.6855726

5 Conclusion

Cuckoo Hashing is an excellent example of **Time and Space Trade off**. The performance of Sequential Cuckoo Hashing is entirely dependent on the search sequence. Data loss depends on "MaxLoop" and data loss decreases with an increase in "MaxLoop". By increasing the "MaxLoop" threshold, the size of **stash memory** may be significantly decreased. The proposed performance indicator, $\frac{T.R.C.}{Key}$, appears as an essential parameter to gauge the **switching cost** in the hashing technique that uses more than one hash table under a single thread model. The proposed performance indicator, **Degree of Dexterity**, quantifies average searching time and insertion latency collectively. This performance indicator is well-suited for the estimation of **cost of first use** of a hash table. Besides **inefficient memory usage**, **high insertion latency**, and **data migration**, this paper identifies one more challenge, **high switching cost** associated with Sequential Cuckoo Hashing. Current work concludes by maintaining a high value of "MaxLoop", execution of costly rehash operations can be avoided in Sequential Cuckoo Hashing.

References

[1] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, and S. Belfkih, "Big Data Technologies: A Survey," *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, pp. 431–448, 2018.

[2] V. Palanisamy and R. Thirunavukarasu, "Implications of Big Data Analytics in Developing Healthcare Frameworks – a Review," *Journal of King Saud University - Computer and Information Sciences*, vol. 31, no. 4, pp. 415–425, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157817302938

[3] S. Mezzoudj, A. Behloul, R. Seghir, and Y. Saadna, "A Parallel Content-Based Image Retrieval System using Spark and Tachyon Frameworks," *Journal of King Saud University - Computer and Information Sciences*, vol. 33, no. 2, pp.

141–149, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157818307146

[4] W. D. Maurer and T. G. Lewis, "Hash Table Methods," *ACM Computing Surveys (CSUR)*, vol. 7, no. 1, pp. 5–19, 1975.

[5] Z. Xie, W. Ding, H. Wang, Y. Xiao, and Z. Liu, "D-Ary Cuckoo Filter: A Space Efficient Data Structure for Set Membership Lookup," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 190–197.

[6] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More Robust Hashing: Cuckoo Hashing with a Stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2009.

[7] W. Li, Z. Cheng, Y. Chen, A. Li, and L. Deng, "Lock-Free Bucketized Cuckoo Hashing," in *European Conference on Parallel Processing*. Springer, 2023, pp. 275–288.

[8] N. Nguyen and P. Tsigas, "Lock-Free Cuckoo Hashing," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 627–636.

[9] Y. Arbitman, M. Naor, and G. Segev, "Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation," in *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, 2010, pp. 787–796.

[10] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592820

[11] B. Minaud and C. Papamanthou, "Generalized Cuckoo Hashing with a Stash, Revisited," *Information Processing Letters*, vol. 181, p. 106356, 2023.

[12] E. Kim and M.-S. Kim, "Cache-Conscious Cuckoo Hashing Method for Multi-Core Cpus," in *FTRA-AIM 2013: FTRA International Conference on Advanced IT, engineering and Management.* FTRA/KITCS, 2013.

[13] R. Pagh and F. F. Rodler, "Cuckoo Hashing," in *European Symposium on Algorithms.* Springer, 2001, pp. 121–133.

[14] P. Pandey, M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, G. Tagliavini, and R. Johnson, "IcebergHT: High Performance Hash Tables Through Stability and Low Associativity," *Proc. ACM Manag. Data*, vol. 1, no. 1, may 2023. [Online]. Available: https://doi.org/10.1145/3588727

[15] H. Zhu, J. Wan, N. Li, Y. Deng, G. He, J. Guo, and L. Zhang, "Odd-Even Hash Algorithm: A Improvement of Cuckoo Hash Algorithm," in *2021 Ninth International Conference on Advanced Cloud and Big Data (CBD)*, 2022, pp. 1–6.

[16] X. Zou, F. Wang, D. Feng, J. Zhu, R. Xiao, and N. Su, "A Write-Optimal and Concurrent Persistent Dynamic Hashing with Radix Tree Assistance," *Journal of Systems Architecture*, vol. 125, p. 102462, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122000522

[17] Q. Yang, H. Huang, J. Zhang, H. Gao, and P. Liu, "A Collaborative Cuckoo Search Algorithm with Modified Operation Mode," *Engineering Applications of Artificial Intelligence*, vol. 121, p. 106006, 2023.

[18] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More Robust Hashing: Cuckoo Hashing with a Stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2010.

[19] S. Pontarelli, P. Reviriego, and M. Mitzenmacher, "EMOMA: Exact Match in One Memory Access," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 11, pp. 2120–2133, 2018.

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

[21] J. Jiang, Y. Yan, M. Zhang, B. Yin, Y. Jiang, T. Yang, X. Li, and T. Wang, "Shifting Hash Table: An Efficient Hash Table with Delicate Summary," in *2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2019, pp. 1–6.

[22] E. Angelino, M. T. Goodrich, M. Mitzenmacher, and J. Thaler, "External-Memory Multimaps," *Algorithmica*, vol. 67, pp. 23–48, 2013.

[23] D. R. Stinson, "Combinatorial Techniques for Universal Hashing," *J. Comput. Syst. Sci.*, vol. 48, no. 2, pp. 337–346, 1994.

[24] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions (Extended Abstract)," in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 106–112.

[25] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, S. Cao, and Y. Guo, "A collision-Mitigation Cuckoo Hashing Scheme for Large-Scale Storage Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 619–632, 2016.

[26] Y. Sun, Y. Hua, Z. Chen, and Y. Guo, "Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems," in *USENIX Annual Technical Conference*, 2019, pp. 329–344.

[27] A. M. Frieze and T. Johansson, "On the Insertion Time of Random Walk Cuckoo Hashing," *Random Struct. Algorithms*, vol. 54, no. 4, pp. 721–729, 2019.

[28] E. Porat and B. Shalem, "A Cuckoo Hashing Variant with Improved Memory Utilization and Insertion Time," in *2012 Data Compression Conference, Snowbird, UT, USA, April 10-12, 2012*,

J. A. Storer and M. W. Marcellin, Eds. IEEE Computer Society, 2012, pp. 347–356.

[29] T. Maier, P. Sanders, and S. Walzer, "Dynamic Space Efficient Hashing," *Algorithmica*, vol. 81, no. 8, pp. 3162–3185, 2019.

[30] S. Walzer, "Load Thresholds for Cuckoo Hashing with Overlapping Blocks," *ACM Trans. Algorithms*, vol. 19, no. 3, may 2023.

[31] L. Devroye and P. Morin, "Cuckoo Hashing: Further Analysis," *Information Processing Letters*, vol. 86, no. 4, pp. 215–219, 2003.

[32] N. N. Shavit and M. P. Herlihy, "Concurrent Extensible Cuckoo Hashing," Feb. 2 2010, US Patent 7657500B2.

[33] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1093–1108.

[34] T. Wang, S. Yang, H. Kimura, G. Swart, and S. Blanas, "Efficient Usage of One-Sided RDMA for Linear Probing," in *Eleventh International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (AMDS'20)*, 2020.

[35] R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets," *Commun. ACM*, vol. 20, no. 11, p. 841–850, nov 1977. [Online]. Available: https://doi.org/10.1145/359863.359887

[36] M. V. Ramakrishna, "Hashing Practice: Analysis of Hashing and Universal Hashing," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 191–199.

[37] G. Sridevi, M. Ramakrishna, and D. Ashoka, "Comprehensive Performance Study of Hashing Functions," *Computer Science Journal of Moldova*, vol. 31, no. 2, pp. 183–199, 2023.

[38] S. Manohar, M. Vignesh, and G. M. Prabhu, "Sensitive Data Transaction using RDS in AWS," *Advances in Science and Technology*, vol. 124, pp. 782–788, 2023.

[39] P. Flajolet, P. Poblete, and A. Viola, "On the Analysis of Linear Probing Hashing," *Algorithmica*, vol. 22, no. 4, pp. 490–515, 1998.

[40] S. Janson, "Asymptotic Distribution for The Cost of Linear Probing Hashing," *Random Struct. Algorithms*, vol. 19, no. 3-4, pp. 438–471, 2001.

[41] R. A. Mugher and N. A. M. Alhammadi, "Performance Evaluation of Quadratic Probing and Random Probing Algorithms in Modeling Hashing Technique," *Journal of Soft Computing and Data Mining*, vol. 3, no. 2, pp. 52–59, 2022.

[42] R. Morris, "Scatter Storage Techniques," *Communications of the ACM*, vol. 11, no. 1, pp. 38–44, 1968.

[43] M. A. Awad, S. Ashkiani, S. D. Porumbescu, M. Farach-Colton, and J. D. Owens, "Analyzing and Implementing GPU Hash Tables," in *2023 Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 2023, pp. 33–50.

[44] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarría-Miranda, J. Mogill, and J. Feo, "Hashing Strategies for The Cray XMT," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[45] Y. Langsam, M. J. Augenstein, and A. M. Tenenbaum, *Data Structures Using C and C++*, 2nd ed. New Delhi: Prentice-Hall of India, 2009.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[47] F. A. Williams, "Handling Identifies as Internal Symbols in Language Processors," *Communications of the ACM*, vol. 2, no. 6, pp. 21–24, 1959.

[48] M. Raab and A. Steger, "Balls into Bins" — A Simple and Tight Analysis," in *Randomization and Approximation Techniques in Computer Science*, ser. Lecture Notes in Computer Science, vol. 1518, M. Luby, J. D. P. Rolim, and M. Serna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 159–170.

[49] K. Rajwar, K. Deep, and S. Das, "An Exhaustive Review of The Metaheuristic Algorithms for Search and Optimization: Taxonomy, Applications, and Open Challenges," *Artificial Intelligence Review*, vol. 56, p. 13187–13257, 2023.

[50] R. Kutzelnigg, "An Improved Version of Cuckoo Hashing: Average Case Analysis of Construction Cost and Search Operations," *Math. Comput. Sci.*, vol. 3, no. 1, pp. 47–60, 2010.

[51] S. Park, M. Kim, and H. Y. Yeom, "GCMA: Guaranteed Contiguous Memory Allocator," *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 390–401, 2018.

[52] Y. Hadjadj, C. M. A. Zouaoui, N. Taleb, S. Mazari, M. El Bahri, and M. Chikr El Mezouar, "VCMalloc: A Virtually Contiguous Memory Allocator," *IEEE Transactions on Computers*, vol. 72, no. 12, pp. 3431–3442, 2023.

[53] C. Gong, C. Tian, Z. Wang, S. Wang, X. Wang, Q. Fu, W. Qin, L. Qian, R. Chen, J. Qi, R. Wang, G. Zhu, C. Yang, W. Zhang, and F. Li, "Tair-PMem: A Fully Durable Non-Volatile Memory Database," *Proc. VLDB Endow*, vol. 15, no. 12, p. 3346–3358, aug 2022. [Online]. Available: https://doi.org/10.14778/3554821.3554827

[54] T. Bozsolik, "Random Numbers," 2019. [Online]. Available: https://www.kaggle.com/dsv/816507

[55] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

Rajeev Ranjan Kumar Tripathi
Pradeep Kumar Singh, Sarvpal Singh

Received May 23, 2024
Revised 1: June 4, 2024
Revised 2: February 12, 2025
Accepted February 26, 2025

Rajeev Ranjan Kumar Tripathi
ORCID: https://orcid.org/0000-0002-8539-6707
Institution: Madan Mohan Malaviya University of Technology, Gorakhpur
Address: Uttar Pradesh, India
E–mail: rajeevranjankumartripathi@gmail.com

Pradeep Kumar Singh
ORCID: https://orcid.org/0000-0002-4250-5264
Institution: Madan Mohan Malaviya University of Technology, Gorakhpur
Address: Uttar Pradesh, India
E–mail: topksingh@gmail.com

Sarvpal Singh
ORCID: https://orcid.org/0009-0008-1965-2064
Institution: Madan Mohan Malaviya University of Technology, Gorakhpur
Address: Uttar Pradesh, India
E–mail: singhsarvpal@gmail.com