Abstract—Cuckoo hashing is an efficient scheme that guarantees worst case constant lookup time. But the insertion delay is significant in cuckoo hashing when the table is highly occupied. This project explores a modification to cuckoo hashing, called CuckooQ. CuckooQ exploits “the power of two choices paradigm” during insertions to reduce the insertion delay in cuckoo hashing, without affecting the lookup time of the existing scheme. When compared to cuckoo hashing, thus, CuckooQ shows a 28 percent speedup on average during insertions. But this comes at the cost of additional rehashes that result in larger tables.

Index Terms—Hashing; Cuckoo Hashing; Searching; Information retrieval

I. INTRODUCTION

Hashing is highly useful for fast data retrieval especially in large-scale distributed systems. The three main performance parameters of any hashing technique are lookup time, update time and space. For any scalable data-centric system, the computational efficiency of these three operations is, thus, of significant importance.

Cuckoo hashing is a hashing scheme that guarantees worst case constant lookup time [1]. This is important because the earlier hashing schemes only guarantee constant lookup time in the average case, but provide no guarantee for the worst case. Also, cuckoo hashing guarantees constant deletion time and amortized constant insertion time, in the worst case. Thus, cuckoo hashing is useful for fast data retrieval.

One drawback of cuckoo hashing, however, is the high insertion delay that occurs in the case of collisions. This is because an existing item needs to be evicted and moved to its alternate location which may cause another item to be displaced and so on, until an empty slot is found in the hash table. This process is called a cuckoo move. The cuckoo moves tend to be longer when the table occupancy is sufficiently high, around 95 percent. The insertion delay becomes significant when multiple threads concurrently access the hash table. This is because the other threads are waiting when one thread has locked some part of the table for an atomic insert operation. This leads to an overall reduced system performance.

The aim of this project is to explore a modification to cuckoo hashing, called CuckooQ. The goal of CuckooQ is to reduce the length of a cuckoo move during insertion, without affecting the lookup time and the memory requirements of the existing cuckoo hashing scheme. For this purpose, it exploits “the power of two choices” paradigm [2] during insertion. Also, given that each bucket has multiple slots, the oldest item in each bucket, in CuckooQ, is the item in the first slot. Hence, the name CuckooQ, because each bucket is like a queue with the oldest item at the head of the queue. Thus, the items are present in the bucket in the order of insertions. Evicting the oldest item in each bucket reduces the length of the cuckoo move, and also requires less memory as compared to the current scheme that uses breadth first search queue while searching for an empty slot during collisions.

As part of this study, thus, the existing open source implementation of cuckoo hashing, called “libcuckoo” [3], is modified to include the scheme presented in CuckooQ. The resulting implementation is verified using the inbuilt test suite that comes with the libcuckoo. Several experiments are conducted to compare CuckooQ with the existing cuckoo hashing implementations. And the time performance evaluations are reported for the insertion, lookup and deletion operations.

The rest of the paper is organized as follows. Section 2 provides an overview of the existing cuckoo hashing schemes including details on concurrency control in cuckoo hashing. Section 3 explains the CuckooQ scheme in detail. Section 4 gives the details of the experiments conducted followed by the results in section 5. Sections 6 concludes the paper with the specific comments.

II. RELATED WORK

The basic cuckoo hashing [1] consists of a hash table which is an array of buckets. Each bucket consists of multiple slots, usually four. The insertion procedure uses two hash functions to compute the appropriate buckets of an item. If both the buckets have an empty slot, the item is allocated to one bucket selected randomly. If one of the buckets has an empty slot, the item is inserted into that bucket. If, however, both the buckets are full, then an item, say x, is randomly selected from one of the two buckets and reallocated to its alternative bucket. This creates an empty slot for inserting the new item. The item x may displace another item, say y, in case if the alternative bucket for x is also full. This process, called cuckoo move, may continue until either all the items are allocated a slot in the hash table or the maximum number of displacements, usually 500, is reached. In the latter case, the basic cuckoo hashing requires a rehash of all the keys in the hash table using new hash functions. Cuckoo moves tend be long and recurring when the table occupancy is sufficiently high. The amortized time complexity of insertion is, however, O(1), if the load factor of the hash table is below 50 percent [1].

Lookup and deletion are fairly straightforward, where lookup requires checking twice the number of slots per bucket,
and deletion requires one more operation after a successful lookup. Thus, the lookup and deletion operations in cuckoo hashing require constant time in the worst case.

The basic cuckoo hashing, however, does not support concurrency. It cannot support read-write simultaneously because an item being read may not be found in any bucket, when it is in the transition state from one bucket to another during a cuckoo move in a concurrent insert operation. This is called a false miss. MemC3 [4] solves this problem by dividing the insertion procedure into two phases, namely, the search phase and the execution phase. During the search phase, a valid cuckoo path is found but no items are displaced. The items are then moved backwards in the execution phase where the last item is first moved to its empty slot, then the second to last item is moved to the newly vacant slot, and so on. This way, during the displacement, each item is present twice in the table and there are no false misses.

MemC3 [4] uses locking striping and optimistic locking to enable concurrent readers and writers. In lock striping, an array of counters, called version counters, is maintained where a version counter $i$ is shared by all the items in the hash table with the hash value $i$. This array of counters is compact (32 KB) and easily fits into a cache. All the counters are initially 0.

Optimistic locking [4] is implemented using the version counters. During insertion, the version counter is incremented by one, before displacing an item, and again after the item is moved to a new location. Thus, an insert operation increments a version counter by two. A lookup operation first verifies the version counter is even. If it is odd, implying a concurrent insert operation on the item is active, the lookup thread waits and retries. Otherwise, the lookup thread proceeds and checks the version counter again after it finishes reading the two relevant buckets. If the two version counters are different, implying the item was modified by a writer thread, and thus, the lookup thread retries.

Further optimizations to the concurrent cuckoo hashing in MemC3 are made by Li et al. in [5]. The size of critical section is reduced during the insert operation so that the search for an empty slot happens outside the critical section which, thus, involves displacing the existing items and inserting the new item into the hash table. This allows multiple threads to search concurrently for their cuckoo paths and acquire a lock only when making the actual updates. Whereas in MemC3, the lock to the hash table is acquired before the insert operation begins which resulted in one big critical section. Furthermore, BFS is used to find an empty slot which reduces the length of the cuckoo path and thus, the time spent in the critical section. The basic cuckoo hashing and MemC3 both used DFS.

III. Solution

Similar to cuckoo hashing, CuckooQ consists of a large table which is an array of buckets. Each bucket has, usually, four slots. And CuckooQ also uses two hash functions which provide two possible locations for inserting an item into the table. When both the buckets have empty slots, in CuckooQ, an item is inserted into the least loaded bucket. This is done according to “the power of two choices” paradigm [2] and helps to achieve uniform distribution of items across the buckets in the table. This further strives to reduce the number of collisions during insertions. This is in comparison to cuckoo hashing, where for the above-mentioned case, one bucket, out of the two possible buckets, is selected randomly. Also, an empty slot is selected randomly, out of all the empty slots in the chosen bucket. Whereas in CuckooQ, the item is inserted into the leftmost empty slot in the chosen bucket. Thus, in CuckooQ, items are present in the buckets in the order of insertion, from left to right. This helps during evictions in the case of collisions, as explained below.

A collision occurs when both the buckets are fully occupied. In this case, an item is evicted and allocated to its alternative bucket. In CuckooQ, the oldest item from the first bucket (obtained from the first hash function) is evicted, whereas in cuckoo hashing, a randomly selected item is evicted from a randomly selected bucket. Since items are present in the buckets in the order of insertion in CuckooQ, as mentioned previously, thus, the oldest item is the item in the first slot in any bucket. After eviction, the items currently in the bucket are shifted left to maintain the order of insertion and thus, the incoming item is allocated to the rightmost slot in the bucket. The newly displaced item is allocated to its alternative bucket which may be full too, thus, causing another item to be evicted and so on, until an empty slot is found in the table. This process is called a cuckoo move, as explained previously, and thus, the eviction in CuckooQ follows the algorithm 1. Figure 1 shows an example of the table before and after insertion in CuckooQ in the case of a collision.

![Eviction in CuckooQ](image)

**Algorithm 1**  

Eviction algorithm in CuckooQ.

Input: An item that needs to be inserted into the hash table.

1: $bucket = h_1(item)$
2: while $bucket$ is full do
3:     $new\_item = left\_Shift(bucket)$
4:     allocate item to $bucket$
5:     item = $new\_item$
6:     $bucket = alternative\_bucket(item)$
7:     allocate item to the leftmost empty slot in $bucket$

The rational behind the eviction algorithm follows directly from the use of “the power of two choices” paradigm during
insertions in CuckooQ. Each item is inserted into a bucket in CuckooQ because its alternative bucket is relatively more loaded. This is true for both the old and the new items in any bucket. This means that, as compared to the older items in the same bucket, there is a higher probability that the alternative buckets of recently inserted items, in a given bucket, are more loaded. Thus, for the oldest item, there is more uncertainty about the load in its alternative bucket because more time has elapsed since “the power of two choices” was performed for this item. Thus, probabilistically, evicting the oldest item should result in shorter cuckoo paths during insertions.

Lookup in CuckooQ is the same as in cuckoo hashing because any item inserted in CuckooQ, will be present in one of the two buckets. Thus, lookup is constant time in the worst case and involves traversing a maximum of \(2 \times b\) slots, where \(b\) is the number of slots in a bucket in the table.

Deletion involves the lookup operation followed by erasing the item from the bucket. But in CuckooQ, deletion further involves shifting left the remaining items. This is because the order of insertion needs to be maintained so that any new item is inserted after any items already present in a bucket. This also means that there are no empty slots in the middle of the buckets in CuckooQ.

Similarly, in the case of a rehash, when the table is doubled in size, there are two cases where every item currently in the table either remains in its old bucket, or is allocated to a new bucket. This implies in CuckooQ, every item is inserted from left to right, in either case, so that there are no empty slots in the middle of the buckets.

For the purposes of implementing CuckooQ, the existing C++ implementation of cuckoo hashing, called “libcuckoo” [3], is modified to implement the algorithms for insertion, lookup, deletion and rehashing in CuckooQ. The resulting implementation is given in [6].

### IV. Experimental Setup

The time performance of CuckooQ is compared with the two existing variations of cuckoo hashing which use the 2-way DFS [1], and a BFS tree [5], respectively, for searching for an empty slot during a cuckoo move. The cuckoo hashing using the BFS scheme for slot search is already implemented in libcuckoo in [3]. For the purposes of experimentation, thus, the cuckoo hashing with the DFS scheme is also implemented in [7].

The original libcuckoo comes bundled with a test suite that contains a test case, called “universal-benchmark”, for benchmarking the performance of various operations including insertions, deletions, lookups, and/or a mixture of the same. The tests are repeatable provided only one thread is used for performing any necessary operations, and if the same seed is used for generating the sequence of items during those operations for insertion, deletion, lookup, etc. Thus, the same test is repeated, using one thread and the same seed, in the three schemes - CuckooQ, BFS based cuckoo hashing, and DFS based cuckoo hashing, for comparing their performances for various operations.

All the experiments are performed on a sufficiently occupied table, \(\approx 96\) percent occupied. Thus, the given test case is modified to first prefill the table to the desired level, and any following operations are timed for the purposes of performance evaluation. Also, the same scheme, i.e., the insertion scheme in CuckooQ, is used to prefill the table before repeating the same experiments for all the three schemes. This is done to ensure that the same table state is presented to all the three schemes, and the resulting experiments can, thus, be used to fairly compare the three schemes. This is especially important for insertions because then only the same keys will cause a cuckoo move in all the three schemes, and the resulting time measurements are appropriate for comparing their performances with respect to the insertion operation.

An example test command used for evaluating the insertion operation is: “universal-benchmark-high-table-occupancy –initial-capacity 21 –prefill 96 –inserts 100 –num-threads 1 –seed X –total-ops 75". This command executes the test case named “universal-benchmark-high-table-occupancy” where the initial size of the table is \(2^{21}\) buckets. The table is prefilled to 96 percent occupancy. The option ‘--inserts 100’ implies all the operations are inserts, and no reads or deletions are performed during this test. And total \(2^{75}\) insertion operations are performed and timed, after the prefill operation is completed.

For evaluating the lookup and deletion operations, thus, the same test command is executed except the options ‘--reads 100’ and ‘--deletes 100’ replace the option of ‘--inserts 100’ in the given command.

### V. Results

Each of the existing cuckoo hashing schemes use a threshold for the number of buckets examined while searching an empty slot during a cuckoo move, and the table is rehashed (by doubling in size) if the threshold is exceeded without finding any empty slot. The threshold is 500 for the length of the cuckoo path using the DFS scheme [1], and five for the depth of the BFS tree [5]. Thus, several insertion experiments are conducted to establish a similar threshold for CuckooQ. When the table is around 96 percent occupied, the maximum cuckoo path lengths observed during the experiments are shown in a histogram in figure 2. As seen in figure 2, the maximum cuckoo path length observed in any experiment is 63, thus, a threshold of 100 buckets is safely used during a cuckoo move in CuckooQ.

To measure the time performance of the insertion operation, total 50 experiments are conducted for each scheme using the test case defined in the previous section. The results are shown in figure 3 where each result is an average of ten iterations. As observed in figure 3, as compared to the other two schemes, CuckooQ shows \(\approx 28\) percent speedup during insertions on average.

Figure 4 shows the final table sizes after performing the \(2^{75}\) insertions in the above-mentioned experiments, where the initial table size was \(2^{21}\) buckets. As observed in figure 4, thus, the table is mostly four times \((2^{21}\text{ buckets})\) the initial
Fig. 2. Histogram of maximum cuckoo path lengths, observed during insertions, in CuckooQ.

Fig. 3. Average insertion times.

Table in CuckooQ whereas in the other two schemes, the table is twice (2^{22} buckets) the size of the initial table. This means that the table was rehashed twice in CuckooQ but only once in the other schemes. The corresponding load factors of the final tables, i.e., the fraction of the tables occupied, in each of the three schemes are shown in figure 5, where the final table in CuckooQ is ≈ 43 percent occupied whereas it is ≈ 85.5 percent occupied in the other two schemes. This is because the same number of items are inserted in all the three schemes but the final table in CuckooQ is double the size of the final table in the other schemes. Also, when the threshold of 500 is used (instead of 100) for the maximum path length of a cuckoo move in CuckooQ, exactly same load factors, as seen with the threshold of 100, are observed for each experiment in CuckooQ. This indicates that increasing the cuckoo path length does not helps in avoiding the additional case causing rehashing in CuckooQ but not in the other two variants of cuckoo hashing.

CuckooQ does not affect the lookup time of the existing scheme, as seen in figure 6. Thus, similar to cuckoo hashing, CuckooQ guarantees worst case constant lookup time.

Deletion in CuckooQ takes longer, ≈ 24 percent slower on average, as compared to the existing BFS based cuckoo hashing, as observed in figure 7. This is expected because CuckooQ involves additional operations of shifting left the remaining items in the buckets after deletion.

On comparing the memory usage of CuckooQ with the existing BFS based scheme, we see that CuckooQ uses no memory for slot search as it always displaces the item at slot 0 of every bucket throughout a cuckoo move. It, however, uses an array to store the cuckoo path, which has length 100 in the worst case. This is in comparison to the existing implementation of cuckoo hashing [3] which searches for an empty slot, as part of a cuckoo move, by creating a BFS tree starting at the two buckets that map to the incoming item [5]. As it dequeues an item from the BFS queue, it looks for an
empty slot in this item’s alternative bucket. If no empty slot is found, then according to this scheme, all the fours slots in the alternative bucket are enqueued into the BFS queue. This process is repeated until an empty slot is found or the BFS tree reaches a maximum depth of five. Thus, in the worst case, the BFS queue will hold $2^{\ast}(4^4+4^5) = 2560$ elements. The cuckoo path is, however, maximum length five in this case. Thus, overall, in the worst case, CuckooQ uses less memory for slot search during cuckoo moves as compared to the existing implementation of cuckoo hashing.

VI. CONCLUSION AND FUTURE WORK

CuckooQ shows a 28 percent speedup on average during insertions when compared to the BFS based and the DFS based cuckoo hashing schemes. But this comes at the cost of additional rehashes that result in larger tables. Deletion operation is slower on average in CuckooQ due to the additional shifting operations needed to maintain the insertion order in every bucket. Similar to cuckoo hashing though, CuckooQ guarantees worst case constant lookup time. Also, without considering the table size after insertions, in the worst case, CuckooQ uses less memory for slot search during cuckoo moves, as compared to the existing BFS based cuckoo hashing scheme.

As a part of future work though, it seems only relevant to explore insertions in CuckooQ where slot search is done from both the buckets during a cuckoo move. Currently, in CuckooQ, slot search is pursued from only one bucket but as observed in the experiments, there is one, or rarely two, additional cases that cause the table to be rehashed multiple times, and the same is not prevented by increasing the maximum cuckoo path length. On the other hand, cuckoo hashing explores more area of the table by starting slot search from the two buckets corresponding to the incoming item, and thus, a high table load factor is consistently maintained throughout all the experiments. However, the approach of empty slot search from two buckets will increase the time to perform insertions in CuckooQ.

ACKNOWLEDGMENT

I would like to thank Dr. Minseok Kwon for continuous guidance and support. Also, I would like to thank anonymous reviewers for suggestions and feedback to improve the paper.

REFERENCES