Lock-Free Bucketized Cuckoo Hashing

Wenhai Li, Zhiling Cheng(✉), Yuan Chen, Ao Li, and Lingfeng Deng

Wuhan University, Wuhan, China
chengzl@whu.edu.cn

Abstract. Concurrent hash tables are one of the fundamental building blocks for cloud computing. In this paper, we introduce lock-free modifications to in-memory bucketized cuckoo hashing. We present a novel concurrent strategy in designing a lock-free hash table, called LFBCH, that paves the way towards scalability and high space efficiency. To the best of our knowledge, this is the first attempt to incorporate lock-free technology into in-memory bucketized cuckoo hashing, while still providing worst-case constant-scale lookup time and extremely high load factor. All of the operations over LFBCH, such as get, put, "kick out" and rehash, are guaranteed to be lock-free, without introducing notorious problems like false miss and duplicated key. The experimental results indicate that under mixed workloads with up to 64 threads, the throughput of LFBCH is 14%–360% higher than other popular concurrent hash tables.

Keywords: buckized cuckoo hashing · lock-free · data structure · multicore · parallel computing

1 Introduction

With the rapid growth of data volume in the Big Data era, the massive amount of data puts increasing pressure on cloud computing systems [1,18]. As a key component of these systems [3,7–9,15], a high-performance hash table is very important for application usability. In step with Moore's law, the improvement in CPU performance has relied on the increase in the number of cores, leading to higher demands for the scalability of hash tables [16]. Consequently, improving the concurrent performance of hash tables on multicore architectures has become a crucial step in designing data-intensive platforms. In practice, the open-addressing hash table is widely used due to its ability to limit the memory usage of the hash table. However, with the increase in application scale, it is challenging to drive concurrent operations on a dense open-addressing hash table.

As an open-addressing hash table, cuckoo hashing was first proposed in 2004 [14]. It utilizes two hash functions to guarantee a constant-time worst-case complexity for the search operation. It introduces a critical step called "kick out", which will be invoked when other keys have occupied both of the positions corresponding to an insertion. The action involves kicking one of the occupying

J. Cano et al. (Eds.): Euro-Par 2023, LNCS 14100, pp. 275–288, 2023.
https://doi.org/10.1007/978-3-031-39698-4_19

keys to its alternative position, creating space for the incoming insertion. This process is similar to that of a cuckoo bird, which always kicks out the eggs of other birds and places its eggs in the nest, hence the name cuckoo hash. If the alternative location of the kicked key is also occupied, a cascade kick is triggered. As the cascade kick could easily form a loop, in general, the load factor of the primitive design cannot exceed 50%. The bucketized cuckoo-hashing scheme introduces multiple slots per bucket for alleviating the kicking loops. This makes the process applicable for use cases with a load factor exceeding 95% [10].

Another essential problem is concurrency. In cuckoo hashing, it could be very difficult to efficiently prevent the kicking process from affecting readers and writers. As a widespread implementation of bucketized cuckoo hashing, libcuckoo [13] conducts a fine-grained locking approach to reduce the blocking overhead of concurrent threads. It can be shown that the throughput of libcuckoo significantly degrades as long as the number of worker threads continuously increases, e.g., with more than 16 worker threads. Lock-free technique [2] has also been applied to cuckoo hashing. Lfcuckoo [13] accelerates the primitive cuckoo hashing using atomic primitives, such as LOAD, STORE, and Compare-and-Swap (CAS). Two correctness issues, i.e., *false miss* and *duplicated key*, have been addressed in the presence of lock-freedom by lfcuckoo. However, the solution can hardly be applied to the bucketized use case, making it impractical.

To implement a concurrent hash table that can efficiently exploit the increasing number of cores, we introduce lock-free techniques to in-memory bucketized cuckoo hashing. We use single-word atomic primitives to optimize concurrent operations over the bucketized data structure, with thorough consideration of the kicking process for cuckoo hashing. We revise lfcuckoo's helper mechanism for the use case with bucketized data structure. For the false miss problem, inspired by hazard pointers [12], we present a mechanism based on hazard hash value that detects the conflicting hash values when performing "kick out". If a search operation gets a miss and detects the hash value derived from its required key conflict with a key being kicked out, it will retry to ensure that it does not return a false miss when the key is present in the hash table. As for the problem of duplicated keys, we generate a snapshot of the target bucket and delete any duplicated keys within it when necessary. In addition to addressing the two issues that affect correctness, we have also presented lock-free lazy rehash which has never been addressed in the previous studies. To address the issue of data hotspots [4], we have also implemented the hotspot detection and adjustment mechanism, improving the performance of the hash table under a highly skewed workload. Ultimately, we implemented lock-free bucketized cuckoo hashing, which is functionally correct, space-efficient, and scalable.

The rest of this paper is organized as follows. Section 2 gives the basic concepts of libcuckoo and lfcuckoo. Section 3 presents the data structure of LFBCH followed by its basic operations. Section 4 shows the implementation details of the lock-free hash table. Section 5 evaluates the hash table based on benchmark workloads. The related works and the conclusion of this paper are given in Sect. 6 and Sect. 7, respectively.

2 Preliminaries

In this section, we present the basic concepts of bucketized cuckoo hashing and the lock-free revision for primitive cuckoo hashing. Problems when driving lock-free operations over a bucketized hash table are discussed.

2.1 Bucketized Cuckoo Hashing

Libcuckoo is a popular implementation of cuckoo hashing that supports lock-based operations using a bucketized hash table. We will use it as an example to illustrate the critical components of a bucketized hash table in the context of cuckoo hashing.

Data Structure. Figure 1 demonstrates the basic structure of libcuckoo by a typical configuration with two cuckoo-hashing functions and a four-way set-associative bucket. A key calculated by two hash functions is mapped to two buckets, each consisting of four set-associative slots. For concurrency, libcuckoo employs a lock strip technique. Each request first acquires locks corresponding to both target buckets before accessing them. Libcuckoo exhibits good scalability when the user requests follow a uniform distribution in their request keys. However, due to the overhead incurred by lock contention, the system performance sharply degrades when the number of worker threads increases on skewed workloads, even in read-only applications.

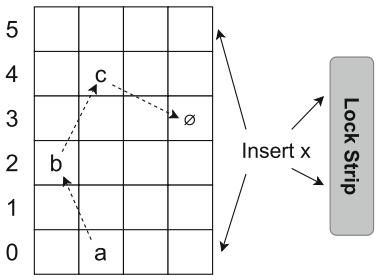
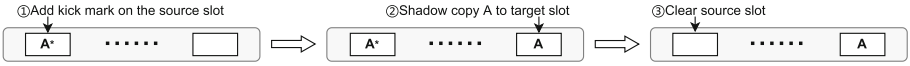


Fig. 1. Demonstration of the kicking process of libcuckoo, where a kick path $a \rightarrow b \rightarrow c \rightarrow \emptyset$ is found to make free slot for a new insertion key "x". $\emptyset$ denotes an empty slot. The three keys, i.e., c, b, a, will be moved to their alternative slots following the directions of the arrows.

Kick Process. The kick process is divided into two stages, i.e., path search and item movement along the kick path. The goal of the path search stage is to find a path for cascade kicking out. For example, Fig. 1 shows how to kick item "a" to make space for a new insertion "x". Libcuckoo finds the kick path by evaluating a BFS search, which guarantees the shortest path [10]. In the second stage, it moves the keys reversely along the kick path. This will leave an empty slot in the head of the kick path, which can be used to accommodate the new insertion key. Rather than locking the entire kicking path, libcuckoo utilizes fine-grained locks to ensure the correctness of the kicking process.

2.2 Difficulties when Supporting Lock-Free Operations

Next, we analyze the critical idea of introducing a lock-free technique into cuckoo hashing. Based on a primitive revision, i.e., lfcuckoo, we highlight the difficulties when considering lock freedom over bucketized cuckoo hashing.



(a) Kicking process: This process involves three modifications to the slot, each of which can be evaluated using the CAS primitive.

(b) False miss: T2 moved Key A while T1 was looking for it, causing T1 to false miss

(c) Duplicated key: T1 and T3 chose to insert the same Key A into different positions, resulting in duplicated key

Fig. 2. The introduction of lock-free techniques into cuckoo hashing can potentially result in two errors: false miss and duplicated key. The gray rounded box in the illustration represents the hash table, and the white boxes indicate the two possible positions of Key A within the hash table. (Color figure online)

Lock-Free Kick. Lfcuckoo is built on a single-slot cuckoo hashing structure, where each hash function corresponds to a single slot. The slots are designed as single words such that atomic primitives can be applied for lock-free purposes. The single item movement in the kicking process is shown in Fig. 2(a). Lfcuckoo marks the least significant bit (LSB) of the source slot pointer at the first step of the kick process. Marking the LSB on the source slot helps to prevent the kick operation from blocking write operations. Other write operations call a helper function to help the "kick out" thread when they detect the kick mark. The helper function encapsulates the processes of slot copy and source clearance, making the kicking process lock-free. While this mechanism makes sense in single-slot cuckoo hashing, it cannot be directly applied to bucketized cuckoo hashing. As multiple target slots in each bucket can be selected as evictee by each movement, a helper thread cannot determine to which slot the marked key should be kicked to.

False Miss. As shown in Fig. 2(b), a false miss refers to the scenario that a key is present in the hash table but a search operation fails to find it. Lfcuckoo resolves false misses by detecting the interleaving kicks based on a kicking counter in the highest 16 bits of the slot. The search operation must be evaluated twice to detect the modifications to the counter on each slot. A false miss might occur if the counter changes within any of the two rounds. It then restarts the search process to make sure whether the key exists. However, using a counter is not entirely safe as there is a risk of fatal errors resulting from short, recycling counters conflicting with each other. Additionally, the presence of version numbers occupies the space available for tags, which is an essential part of reducing memory access and speeding up searches. On the other hand, version numbers are also inapplicable to bucketized cuckoo hashing due to the expansion of slot numbers.

Duplicated Key. Figure 2(c) shows an example of the duplicated key caused by three interleaved modifications. To address this issue, at any time a duplicated key is found by a search process, lfcuckoo always removes the key in the later slot of the search sequence. The solution of lfcuckoo makes sense since each key has only two possible locations, both cannot be selected as evictees due to the duplicate key in their alternative slot. However, in the bucketized cuckoo hashing, a kicking process may be interleaved with the duplicated key check process. It thus might cause the duplicated key check process to miss duplicated keys or to get an intermediate state of the kicking out process.

3 Overview of LFBCH

In this section, we provide an overview of our bucketized cuckoo hashing, including its data structure and fundamental operations.

3.1 Data Structure

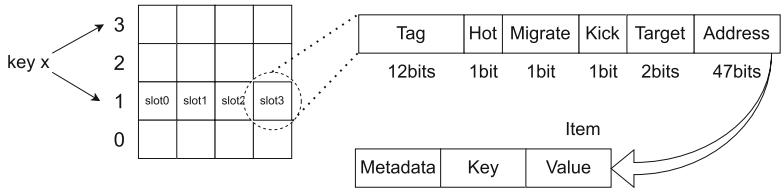


Fig. 3. Data structure of LFBCH with the fine-grained division of its 64-bit slot especially for supporting lock-free operations.

The basic structure of LFBCH is an array-typed bucketized hash table with two hash functions and four slots per bucket. Each slot is 64 bits wide and can be manipulated by atomic primitives. The atomic LOAD result of the slot is referred to as an entry.

As shown in Fig. 3, the entry can be divided into the following fields:

- *Address.* A 47-bit address is generally sufficient to locate a key-value pair for purposes of alignment.
- *Target.* Used to identify the target slot index of an in-flight entry.
- *Kick/Migrate.* Mark that the entry is being kicked/rehash migrated.
- *Hot.* Identify whether an item is frequently accessed.
- *Tag.* A signature of the hash value for each key. Enhances query efficiency by filtering out memory accesses to keys with different signatures.

The compact bucket structure of LFBCH is similar to that of libcuckoo, enabling it to support a load factor of up to 95%. Additionally, with the introduction of lock-free techniques, LFBCH offers significantly better scalability than libcuckoo, which employs a lock strip for synchronization. Next, we show how to drive lock-free operations based on the data structures.

3.2 Basic Operations

In this section, we consider three basic operations, i.e., Get, Put, and Delete. We focus on the use cases without the kicking and rehash processes and leave more details of the two processes in Sect. 4.

Get. Given two distinct hash functions, we can determine two target buckets based on the two hash values of a Get key. We refer to the two hash functions as the primary hash function and the secondary hash function, and the buckets they map to as the primary bucket and the secondary bucket, respectively. Searches always start from the primary bucket and traverse all eight slots across both buckets. It in turn considers each slot by triggering an atomic LOAD to obtain the entry thereon. If all of the eight comparisons have failed, a result of a miss will return. It is worth noting that an interleaving kicking process may issue false misses, as demonstrated in Fig. 2(b). We will detail the resolution in Algorithm 1.

Put. The semantic of the put operation is inserting when the key is missing and updating when the key is hit. We will discuss the two cases separately. For an update operation, only one slot returned from the search process is considered. Two update strategies are employed. For items whose value length is less than 8 bytes, an in-place update is performed by directly updating the value through a CAS operation in the value field of the item. For items with larger values, a Read-Copy-Update (RCU) operation is employed. A new item containing the new key-value pair is created, and then a CAS operation is used to replace the old item with the new item. For insertion, if the search process can find an empty slot, we can employ the *RCU*-based update to insert the new item. If no empty slot is found, the kick-out algorithm kicks out a key within the target buckets to make room for the insertion. After the kick-out algorithm finishes, an empty slot will appear within the target bucket, and we can perform insertion on the empty slot. After the insertion, the action to check and resolve the duplicated key starts. The details are described in Sect. 4.2. Note that if we find a kick mark on the entry of the target key during the search process. We need to call the helper function to help kick out and re-execute the PUT operation from the beginning. The details are covered in Sect. 4.1.

Delete. The Delete operation has similar logic to the Put. If the key is not found, a failure will be returned. Otherwise, we perform a CAS operation to replace the target slot with an empty entry atomically.

4 Detailed Algorithm Description

This section will detail the critical designs in lock-free bucketized cuckoo hashing, including lock-free kicking, preventing duplicated key, lock-free rehashing, and hotspot perception.

4.1 Lock-Free Kick on Bucketized Cuckoo Hashing

Our lock-free kicking algorithm has two primary components: path search and item movement along the kick path.

Path Search. Regarding path search, we have adopted the BFS algorithm employed by libcuckoo without locking. However, since other threads may modify the state of the hash table during the search process, the state of the hash table may be modified by other threads during the path search process. For example, the empty slot at the tail may be filled with a key after the search path is formed. Any inconsistencies between the actual state and the kick path are checked for in the following item movement along the kick path phase. If any discrepancies are found, the kicking-out process is restarted from the beginning to ensure correctness.

Item Movement Along Kick Path. Once a kick path is found, the items need to be moved along the kick path. Proceeding from the tail towards the head, the process moves one item at a time. The fundamental operations of the single-item movement process are similar to those performed by lfcuckoo and can be referred to in Fig. 2(a). We have added our method to prevent false misses and improve the handling of bucketized environments. The specific method is shown in Algorithm 1.

Meanings of the key variables adopted by the algorithm are as follows: The *table* represents the entire hash table. The *source_bucket* and *source_slot* represent the bucket and source slot indexes. We abbreviate the two variables as *sb* and *ss*. Similarly, *target_bucket* and *target_slot* are respectively abbreviated as *tb* and *ts*. The *source_entry* represents the value of the uint64_t variable maintained on the source slot. The *kick_marked_entry* is the kick-marked result of source_entry.

Two global arrays are defined (lines 1–2) with lengths equal to the number of global threads. Each thread is mapped to a specific position in the array according to its thread id. Padding is used to avoid false sharing issues. The functions of these global arrays are: *hash_record.* The working thread stores the hazard hash value of the key calculated by the primary hash function in the corresponding position of the hash record array at the beginning of every operation, for conflict detection performed by the kicking threads; *retry_flags.* Once the hazard hash value of another worker thread is found to conflict with the moving item, the retry_flag at the corresponding position of the reader will be set, indicating this worker thread might be affected by movement.

The item_move function (line 4) begins with two initial checks. The first "if" statement (line 6) checks if the source slot is empty. If it is empty, the item_move function can return success directly. The second "if" statement (line 8) checks if the source slot has been marked with a "kick mark" by other threads, indicating that another thread is concurrently accessing it for an item move operation. A helper function is then invoked to help the moving process and prevent blocking. The target slot information has been added to the kick_marked_entry (line 11) for a potential helper to obtain (line 18). Otherwise, other threads cannot determine which slot they should help kick into. The helper will get the target bucket information by calculating the two possible bucket locations of the intended key based on the item associated with the kick_marked_entry. The bucket that differs from the source bucket is identified as the target bucket.

Algorithm 1. Lock Free Single Item Movement

```
1: atomic < bool > retry_flags[thread_num]
2: atomic < uint64_t > hash_record[thread_num]
3:
4: function single_item_move(sb, ss, tb, ts)
5:     source_entry ← table[sb][ss].LOAD()
6:     if source_entry == empty_entry then
7:         return true
8:     if is_kick_marked(source_entry) then
9:         helper(sb, ss, source_entry)                    ▷ source entry here is kick marked
10:        return false
11:    kick_marked_entry ← source_entry, ts
12:    if !table[sb][ss].CAS(source_entry, kick_marked_entry) then
13:        return false
14:    return copy(sb, ss, tb, ts, kick_marked_entry)
15:
16: function helper(sb, ss, kick_marked_entry)
17:    key, hash ← kick_marked_entry
18:    tb ← hash, sb; ts ← kick_marked_entry
19:    copy(sb, ss, tb, ts, kick_marked_entry)
20:
21: function copy(sb, ss, tb, ts, kick_marked_entry)
22:    if table[tb][ts].CAS(empty_entry, source_entry) then
23:        hash ← source_entry
24:        set_retry_if_hazard(hash)
25:        if table[sb][ss].CAS(kick_marked_entry, empty_entry) then
26:            return true
27:    if key_in(ss, sb) == key_in(ts, tb) then
28:        hash ← source_entry
29:        set_retry_if_hazard(hash)
30:        table[sb][ss].CAS(kick_marked_entry, empty_entry)
31:        return false
32:    table[sb][ss].CAS(kick_marked_entry, source_entry)
33:    return false
34:
35: function set_retry_if_hazard(hash)
36:    for i = 0 → thread_num − 1 do
37:        if hash_record[i].LOAD() == hash then           ▷ Check hazard value
38:            retry_flags[i].STORE(true)
39:
40: function search(key)
41:    hash_record[thread_id].STORE(hash)                  ▷ Store hazard value
42:    while true do
43:        bool hit ← search_two_buckets(key)
44:        if hit then
45:            return key_hit
46:        else if retry_flags[thread_id].LOAD() then
47:            retry_flags[thread_id].STORE(false)
48:            continue
49:        else
50:            return key_miss
51:
```

In the copy function (line 21), a CAS operation is first used to update the target slot with the source entry (line 22). If the CAS operation succeeds, a CAS operation is then used to clear the source slot (line 25). If this clearing operation succeeds, the copy operation is considered successful and the function returns true.

Before clearing the source slot (line 25), the function set_retry_if_hazard is invoked to prevent other threads from returning false misses that may have been

affected by the item movement process. This function (line 35) traverses the hash_record array to determine if the hash value of the key being moved conflicts with the hazard hash value of a key being operated on by another thread. If it is, the retry_flag of the corresponding thread is set, informing other threads that the search may have been affected by the item movement process and a false miss may have occurred. To avoid false misses, the search algorithm (line 40) begins by storing the hash value of the target key in the corresponding position of the hash_record array. If it gets a miss and the corresponding retry_flag is set, it indicates that the miss may be a false miss. In this case, the retry flag is cleared, and the search operation is performed again.

If either of the two CAS operations (line 22, line 25) fails, it indicates that the state of the hash table has been modified by another thread, and failure handling is required. Firstly, it is necessary to check whether the keys in the target and source positions are the same. If they are the same, it means that another thread has already completed the entry copying operation. In this case, the source slot needs to be cleared (lines 28–30), and the function returns false. If the keys are not the same, it means that either the target slot or the source slot has already been modified by another thread. In either case, the copy operation cannot succeed. At this point, a CAS operation is used to attempt to clear the kick-out mark in the source slot (line 32) and restore it to its state before the mark was set. The function then returns false.

4.2 Prevent Duplicated Key

Unlike the temporary duplicated keys that may arise during item movement, the presence of duplicated keys resulting from distinct threads inserting the same key into different empty slots can cause errors in the hash table. We conduct a post-checking step after each insertion to solve this problem.

The specific methodology is shown in Algorithm 2. The *check_duplicate_key* function is called after each successful insertion. The duplicated key check will pass only when the number of target keys in the obtained snapshot equals 1. Since in the vast majority of cases, post-checking only scans buckets that have already been scanned during the search phase and passes the check without introducing additional overhead, our post-checking mechanism is efficient.

4.3 Lock Free Rehash

When the load factor of the hash table is excessively high, rehash is necessary. Similar to that of libcuckoo, our rehash mechanism employs a lazy rehash strategy, whereby items are shallow-copied gradually from the old table to the newly created table. However, unlike libcuckoo, we use a global atomic bitmap to identify whether each bucket has been migrated. Moreover, for each item migration from the old table, we use a mechanism similar to the single item movement within a table, but with migrate marks instead of kick marks. Therefore, the guarantee of lock-free rehashing is ensured.

Algorithm 2. Post-checking Process for Duplicated Key

```
1: //b : bucket_index, s : slot_index
2:
3: function check_duplicate_key(key)
4:     Start :
5:     Initialize snapshot
6:     for each slot in two buckets do
7:         snapshot.append(⟨b, s, slot.LOAD()⟩)                ▷ 8 slots in total
8:     if retry_flag[thread_id].LOAD() then
9:         retry_flag[thread_id].STORE(false)
10:        goto Start
11:    count = 0
12:    for ⟨b, s, entry⟩ in snapshot do
13:        if is_kick_marked(entry) then
14:            helper(b, s, entry)
15:            goto Start
16:        key_extract ← entry
17:        if key equals key_extract then
18:            task ← b, s, entry
19:            count++
20:    if count ≤ 1 then
21:        return
22:    else
23:        table[task.b][task.s].CAS(entry, empty_entry)
24:        goto Start
```

4.4 Hot Key Perception and Adjustment

Although bucketized cuckoo hashing minimizes memory access and cache misses due to its compact slot layout, queries on keys in the secondary bucket result in one additional cache miss compared to the primary bucket. Additionally, due to skewed key distribution in practice, when a hotspot key is placed in the secondary bucket, accessing it incurs additional overhead. Therefore, we have optimized our implementation by placing hotspot keys as far forward as possible in the primary bucket to reduce the number of comparisons required for access.

The specific method of adjusting hotspots is to displace the first non-hotspot key located before the hotspot key in the search sequence and subsequently place itself in the vacated position. The process of displacing a key is the same as the kicking process, except that there is no cascading displacement. If the secondary bucket of a non-hotspot key is full, it is skipped. Hotspot keys are determined based on whether the slot has a "hot" mark, which is applied the first time the key is updated. Considering the scenarios with their hotspots frequently evolving, the "hot" marks on all the slots in the bucket will be cleared after each successful adjustment of a hotspot key. If all the keys before the search sequence of a hotspot key are hotspot keys, no adjustment is performed.

5 Experiments

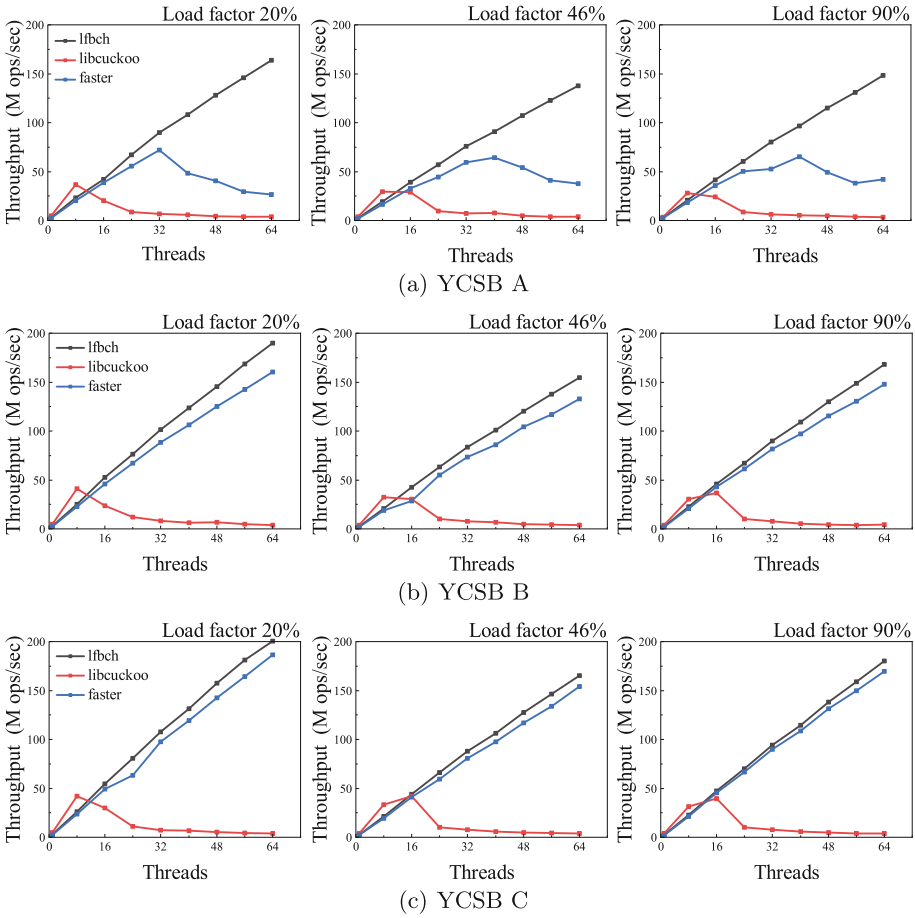
In this section, we evaluate the performance of LFBCH using YCSB benchmarks. We compare the throughput and scalability of LFBCH with that of libcuckoo and the hash table faster [3] use. We also provide the results with the hotspot optimization.

Environment. We conducted experiments on a machine comprising two AMD EPYC 7742 64-Core Processors with 1.50 GHz processors. Each processor has two sockets, each with 64 cores. The RAM capacity is 1024 GB. It runs Ubuntu 16.04.7 LTS OS with Linux 4.4 kernel. Only 64 cores in one socket were utilized, and each thread was bound to a specific core. Jemalloc library is used to allocate memory. All code is compiled using gcc/g++−7.5.0 with parameter −O2. Memory reclamation is not carried out to eliminate the influence of different memory reclamation algorithms on the hash table throughput.

Table 1. Load factor within different slot number

Slot Per Bucket	1	2	4	8
Load Factor	50%	87%	95%	95%

BaseLine and Workloads. We employed libcuckoo and the hash table faster uses in contrast to LFBCH. As shown in Table 1, since each bucket in lfcuckoo



(a) YCSB A

(b) YCSB B

(c) YCSB C

Fig. 4. The throughput under different load factors and different YCSB loads varies with the worker threads.

only has a single slot, the load factor cannot exceed 50%, making it impractical and thus not included in the comparison experiments. In each test, the number of buckets for Libcuckoo and LFBCH was set at 2^{27}. Because of the batch chain structure of the faster hash table, it has half as many buckets. We conduct experiments using the YCSB core workloads A, B, and C [6]. The Zipfian distribution parameter in YCSB is set to 0.99. For each item, we set both the key size and the value size to be 8 bytes.

Results. The throughput test result is shown in Fig. 4. We observed that the throughput of libcuckoo decreases when the number of threads exceeds 16 in all cases. At 64 threads, the throughput is less than five million requests per second. In contrast, the throughput of LFBCH increases linearly for all thread counts. When the load type is the same, the performance of each hash table is better when the load factor is low, compared to when the load factor is high.

We focus on the situation with a load factor of 46% to reflect general conditions. Under YCSB A load, LFBCH achieved the highest throughput of 148 million requests per second at 64 threads, while faster's throughput was only 41 million requests per second. Faster and libcuckoo achieved their highest throughputs at 40 and 8 threads respectively, but LFBCH was still 127% and 428% higher than their highest throughputs, respectively. The reason for the performance degradation of faster is the result of RCU update contention. Under YCSB B load, LFBCH had the highest throughput of 168 million requests per second, 14% higher than faster's highest throughput and 360% higher than libcuckoo's highest throughput. Under YCSB C load, LFBCH had the highest throughput of 18 million requests per second, 6% higher than faster's highest throughput and 350% higher than libcuckoo's highest throughput. Faster and LFBCH perform similarly under YCSB C loads because faster uses the same atomic load and tag acceleration mechanisms for search as LFBCH.

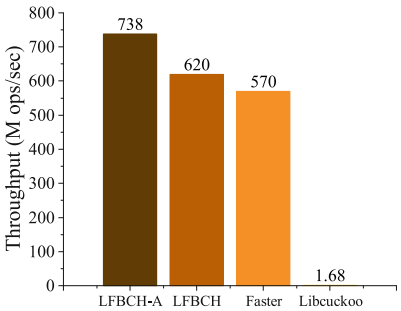


Fig. 5. Hotspot adjustment result

We used YCSB C load with a Zipf coefficient of 1.22 to experiment with our hotspot adjustment strategy in skew workload. LFBCH-A represents the results obtained after enabling the hotspot adjustment. As shown in Fig. 5, The performance of LFBCH improved by 19% after hotspot adjustment and optimization.

6 Related Works

Various hash tables based on cuckoo hashing have been widely addressed. The primitive cuckoo hash algorithm was first proposed in 2004 [14] in which each key is mapped to two positions using two hash functions, and insertion is guaranteed by the use of a "kick out" operation. The primitive version has a low load factor and no concurrency scheme.

Memc3 [8] allows multiple readers and a single writer concurrently access the bucketized cuckoo hashing. Xiaozhou Li's work [10] introduced HTM into bucketized cuckoo hashing. It has inherent limitations of HTM, which can lead to significant performance degradation when transactions fail frequently. As a widely used cuckoo-based hash table, libcuckoo [11] employs a bucketized structure and utilizes fine-grained locks to control concurrent access. However, the performance degradation caused by the competition when the number of threads increases cannot be avoided. Lfcuckoo [13] made a Lock-free improvement to the primitive cuckoo hashing, but it has not been extended to bucketized cuckoo hashing, leading to low space utilization efficiency. Level hash [5] has also employed a lock-free technique in bucketized cuckoo hashing, but it is designed for the scenario of persistent memory.

The work of hotspot adjustment for our hash table is inspired by hotring [4]. It speeds up the performance of the chained hash table in the case of data skew by pointing the linked list header pointer to the hotspot key.

7 Conclusion

We introduced lock-free techniques into bucketized cuckoo hashing and proposed LFBCH, which paves the way toward scalability and high space efficiency. All of the operations over LFBCH are guaranteed to be lock-free, without introducing notorious problems like false miss and duplicated key. Lock-free rehash and hotspot adjustment are also implemented in LFBCH. The throughput of LFBCH is 14%–360% higher than other popular concurrent hash tables.

Accessing the same shared data structure introduces additional latency when the working threads are distributed across CPU sockets. In the future, optimizations can be made to improve the performance of LFBCH in scenarios where it is distributed across CPU sockets.

Acknowledgment and Data Availability Statement. This work was sponsored by the National Science Foundation, grant 61572373. Besides the reviewers, we would like to thank High Performance Computing Center at the Computer School of Wuhan University. The Code is available in the Figshare repository: https://doi.org/10.6084/m9.figshare.23550111 [17]

References

1. Armbrust, M., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
2. Barnes, G.: A method for implementing lock-free shared-data structures. In: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 261–270 (1993)
3. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., Barnett, M.: Faster: a concurrent key-value store with in-place updates. In: Proceedings of the 2018 International Conference on Management of Data, pp. 275–290 (2018)

4. Chen, J., et al.: Hotring: a hotspot-aware in-memory key-value store. In: FAST, pp. 239–252 (2020)
5. Chen, Z., Hua, Y., Ding, B., Zuo, P.: Lock-free concurrent level hashing for persistent memory. In: Proceedings of the 2020 USENIX Conference on USENIX Annual Technical Conference, pp. 799–812 (2020)
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)
7. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation techniques for main memory database systems. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 1–8 (1984)
8. Fan, B., Andersen, D.G., Kaminsky, M.: MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. In: Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13), pp. 371–384 (2013)
9. Garcia-Molina, H., Salem, K.: Main memory database systems: an overview. IEEE Trans. Knowl. Data Eng. **4**(6), 509–516 (1992)
10. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the Ninth European Conference on Computer Systems, pp. 1–14 (2014)
11. M, K.: libcuckoo. https://github.com/efficient/libcuckoo
12. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
13. Nguyen, N., Tsigas, P.: Lock-free cuckoo hashing. In: 2014 IEEE 34th International Conference on Distributed Computing Systems, pp. 627–636. IEEE (2014)
14. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
15. Ghemawat, S., Dean, D.: levelDB. https://github.com/google/leveldb
16. Shalf, J.: The future of computing beyond Moore's law. Phil. Trans. R. Soc. A **378**(2166), 20190061 (2020)
17. Li, W., Cheng, Z., Chen, Y., Li, A., Deng, L.: Artifact and instructions to generate experimental results for euro-par 23 paper: Lock-free bucketized cuckoo hashing (2023). https://doi.org/10.6084/m9.figshare.23550111
18. Wu, S., Li, F., Mehrotra, S., Ooi, B.C.: Query optimization for massively parallel data processing. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, pp. 1–13 (2011)