Lock-free Cuckoo Hashing

Nhan Nguyen, Philippas Tsigas
Chalmers University of Technology
Gothenburg, Sweden
Email: {nhann, tsigas}@chalmers.se

***Abstract*—This paper presents a lock-free cuckoo hashing algorithm; to the best of our knowledge this is the first lock-free cuckoo hashing in the literature. The algorithm allows mutating operations to operate concurrently with query ones and requires only single word compare-and-swap primitives. Query of items can operate concurrently with others mutating operations, thanks to the two-round query protocol enhanced with a logical clock technique. When an insertion triggers a sequence of key displacements, instead of locking the whole cuckoo path, our algorithm breaks down the chain of relocations into several single relocations which can be executed independently and concurrently with other operations. A fine tuned synchronization and a helping mechanism for relocation are designed. The mechanisms allow high concurrency and provide progress guarantees for the data structure's operations. Our experimental results show that our lock-free cuckoo hashing performs consistently better than two efficient lock-based hashing algorithms, the chained and the hopscotch hash-map, in different access pattern scenarios.**

I. Overview

A hash table is a fundamental data structure which offers rapid storage and retrieval operations. Hash tables are widely used in many computer systems and applications. Papers in the literature have studied several hashing schemes which differ mainly in their methods to resolve hash conflicts. As multi-core computers become ubiquitous, many works have also targeted the parallelization of hash tables to achieve high performance and scalable concurrent ones.

Cuckoo hashing [1] is an open address hashing scheme which has a simple conflict resolution. It uses two hash tables that correspond to two hash functions. A key is stored in one of the tables but not in both. The addition of a new key is made to the first hash table using the first hash function. If a collision occurs, the key currently occupying the position is "kicked out", leaving the space for the new key. The "nestless" key is then hashed by the second function and is inserted to the second table. The insertion process continues until no key is "nestless". Searching for a key involves examining two possible slots in two tables. Deletion is performed in the table where the key is stored. Search and delete operations in cuckoo hashing have constant worst case cost. Meanwhile, insertion operations with the cuckoo approach have been also proven to work well in practice. Cuckoo hashing has been shown to be very efficient for small hash tables on modern processors [2].

In cuckoo hashing, the sequence of the evicted keys is usually referred to as "cuckoo path". It might happen that the process of key evictions is a loop, causing the insertion to fail. If this happens, the table needs to be expanded or rehashed with two new hash functions. The probability of such insertion failure is low when the load factor[1] is lower than 0.49 but increases significantly beyond that [1]. Recent improvements address this issue by either using more hash functions [3] or storing more than one key in a bucket - known as *bucketized cuckoo hashing* [4] [5].

A great effort has been made to build high performance concurrent hash tables running on multi-core systems. Lea's hash table from Java Concurrency Package [6] is an efficient one. It is a closed address hash table based on chain hashing and uses a small number of locks to synchronize concurrent accesses. Hopscotch hashing [7] is an open address algorithm which combines linear probing with the cuckoo hashing technique. It offers a constant worst case look-up but insertion might requires a sequence of relocation similar to the cuckoo hashing. The concurrent hopscotch hashing synchronizes concurrent accesses using locks, one per bucket. A concurrent version of cuckoo hashing found in [8] is a bucketized cuckoo hash table using a stripe of locks for synchronization. As lock-free programming has been proved to achieve high performance and scalability [9] [10], a number of lock-free hash tables have also been introduced in the literature. Micheal, M. [11] presented an efficient lock-free hash table with separated chaining using linked lists. Shalev O. and Shavit N. [12] designed another high performance lock-free closed address resizable hash table. In [13], a lock-free/wait-free hash table is introduced, which does not physically delete an item. Instead, all the live items are moved to a new table when the table is full.

To the the best of our knowledge, there has not been any lock-free cuckoo hashing introduced in the literature. There are several reasons which can explain this fact. Because a key can be stored in two possible independent slots in two tables, synchronization of different operations becomes a hurdle to overcome when using lock-freedom. As an example, two insertion operations of a key with different data can simultaneously and independently succeed; this can

[1] Load factor: the ratio between the total number of elements currently in the table over its capacity.

cause both of them to co-exist, which is not aligned with the common semantics of hash tables in the literature. In addition, a relocation of a key from one table to another is a combination of one remove and one insert operations, which need to be combined in a lock-free way. While taking care of that, the relocation of a key when it is being looked up can cause the look-up operation to miss the key, though it is just relocated between tables.

In this work, we address these challenges and present a lock-free cuckoo hashing algorithm. We do not consider bucketized cuckoo hashing. To the best of our knowledge, this is the first lock-free cuckoo hashing algorithm in the literature. Our algorithm tolerates any number of process failures. The algorithm offers very high query throughput by optimizing the synchronization between look-up and modification operations. Concurrency among insertions is also high thanks to a carefully designed relocation operation. The sequence of relocations during insertion is broken down into several single relocations to allow higher concurrency among operations. In addition, a fine tuned helping mechanism for relocation operations is designed to guarantee progress. Our evaluation results show that the new cuckoo hashing outperforms the state-of-the-art hopscotch and lock-based chained hash tables [14].

The rest of this paper is organized as follows. Section II introduces our algorithm in a nutshell. The full design together with a pseudo-code description is presented in Section III. Section IV provides the proof of correctness of the algorithm. Experiments and evaluation results are presented in V. Finally, Section VI concludes our paper.

II. Lock-free cuckoo hashing algorithm

Our concurrent cuckoo hashing contains two hash tables, hereafter called sub-tables, which correspond to two independent hash functions. Each key can be stored at one of its two possible positions, one in each sub-table. To distinguish the two sub-tables, we refer to one as the primary and to the other as the secondary. The look-up operation always starts searching in the primary sub-table and then in the secondary one. Because there are different use cases of looking-up operations, we divide them into two types. One, *search*, is a query-only one which asks for the existence of a key without modifying the hash table. The other one is a query as a part of another operation such as a deletion or an insertion. We refer to it as *find* to distinguish it from the "real" *search*.

A *search* operation starts by examining the possible slots in the primary sub-table first, and then in the secondary one, and reports if the searched key is found in one of them. Such a simple search, however, can miss an existing key and report the key as not found. The reason is that the reading from two slots is not performed in one atomic step and a relocation operation might interleave in between. The searched key can be relocated from the secondary to the primary sub-table but is missed by the above reading operations. We called such key a "moving key". To deal with this issue, we design a two round query protocol enhanced with a logical clock based counter technique. Each hash table slot has a counter attached to it to count the number of relocations at the slot. The first round of the two round query reads from the two possible slots and check for the existence of the searched key like the mentioned simple search does. In addition, it records the slot's counter values. If the key is not found, the second round does similar readings and examination. The second round can discover the key if it was relocated from the secondary to the primary sub-table, and was missed by the first round query. However, it might also miss the key if it has been relocated back and forth between sub-tables several times and interleaved with the readings. Therefore, the second round also records the counter values and compares them with the values of the first round. If the new values are at least two units higher than the previous ones, there is a possibility that even the two round query misses the key because two or more interleaving relocations have happened. In this case, the search is reexecuted.

The *insert* operation of a key starts by invoking *find* to examine if the key exists. If it does not, the insertion is made to the primary sub-table first and, only if a collision occurs, to the secondary sub-table. If both positions are occupied, a relocation process is triggered to displace the existing key to make space for the new key. The original cuckoo approach [1] inserts the new key to the primary sub-table by evicting a collided key and re-inserting it to the other sub-table, as described in Section I. This approach, however, causes the evicted key to be "nestless", i.e. absent from both sub-tables, until the re-insertion is completed. This might be an issue in concurrent environments: the "nestless" key is unreachable by other concurrent operations which want to operate on it. One way to deal with this issue is to make the whole relocation process atomic, which is not efficient and scalable since it is going to result in coarse grained synchronization. We approach the relocation process differently. If an insertion requires relocation, an empty slot is created before the new key is actually added to the table. Our approach contains two steps. First, we search for the cuckoo path, to find a vacant slot. Thereafter, the vacant slot is "moved" backwards to the beginning of the cuckoo path by "swapping" with the last key in the cuckoo path, then the second last key and so forth. The new key is then inserted to the empty slot using an atomic primitive. Each "swap" step involves modifications of two slots in two sub-tables. We can design a fine tuned synchronization for the "swap" using single word Compare-And-Swap (CAS)[2] primitives by using the pointer's Least-Significant-Bit (LSB) marking technique. This technique, which takes advantages of aligned memory addresses and is widely used in the literature,

[2] CAS is a synchronization primitive available in most modern processors. It compares the content of a memory word to a given value and, only if they are the same, modifies the content of that word to a given new value.

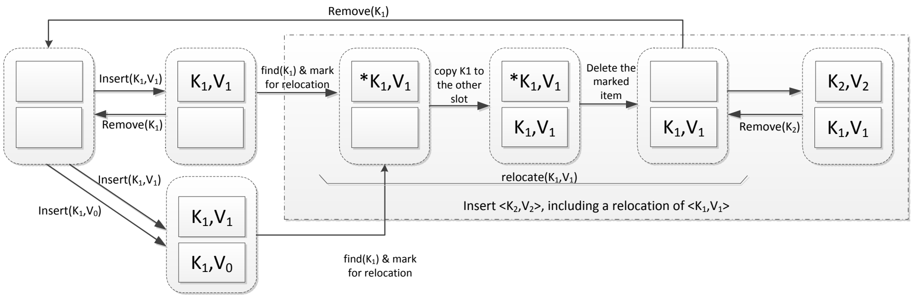

Figure 1: State transition of two possible positions of a key in the primary (upper) and secondary sub-tables

set the unused LSB for certain purposes. In our case, it indicates one thread's intention to relocate a table's entry. A helping mechanism is also designed so that other concurrent operations can help finishing an on-going relocation.

Another issue with insertion operations is that a key can be inserted to two sub-tables simultaneously. Since concurrent insertions can operate independently on two sub-tables, they can both succeed. This results in two existing instances of one key, possibly mapped to different values. To prevent such a case to happen, one can use a lock during insertion to lock both slots so that only one instance of a key is successfully added to the table at a time. As we aim for high concurrency and strong progress guarantees, we solve this problem differently. Our table allows, in some special cases, two instances of a key to co-exist in two sub-tables. The special cases are when the two instances have been inserted to two sub-tables simultaneously. (It is noticed that, concurrent insertions to the same sub-table operate and linearize normally, which guarantees that only one instance of a key exists in one sub-table). We design a mechanism to delete one of the instances internally and as soon as possible so that our table still provides the conventional semantic in which one key is mapped to one value. The question is: which instance between the two is to be deleted? Since the two successful insertions which lead to the co-existence must be concurrent, it is always possible to order them so that the insertion to the secondary sub-table is linearized before the insertion to the primary one. In this way, the latter insertion "overwrites" the data inserted by the former one. As a result, if the same key is found in both sub-tables at a certain time, the key in the primary table is the only valid one. Our mechanism to realize such duplication and remove the overwritten key, i.e the one in the secondary sub-table, will be described after discussing below the consequence of such co-existence to the hash table's operations.

We now analyze the consequence of the co-existence of two instances of a key to the operations of the hash table, beginning with the look-up operations. The query-only *search* first examines the primary hash table, and can report if the searched key is stored there immediately. In the cases that two instances of a key co-exist in two sub-tables, *search* always returns the one in the primary sub-table, which agrees with the semantics described in the previous paragraph. The *find* operation, on the other hand, is used at the beginning of any *insert* or *remove* operation. The result of the invoked *find*, i.e the key exists in one or both sub-tables, affects the way the invoker behaves. Therefore, if *find* discovers that two instances of a key exist, it deletes the one in the secondary sub-table, as being described in detail in the next section. When an *insert* or a *remove* proceeds after *find* returns, there is going to be one instance of the key in the table.

The remove operation of a key starts by invoking *find* to locate the table's slot where the key is being stored. Then the key is removed by means of a CAS primitive.

Figure 1 shows the states of two possible positions on two sub-tables where a key K_1 can be hashed to. The states change according to the operations performed. The operations are, for example, an *insert*(K_1, V_1), a *remove*(K_1, V_1), two concurrent *insert*(K_1, V_0) and *insert*(K_1, V_1), and an *insert*(K_2, V_2) (in the dashed rectangle) which requires a relocation of $< K_1, V_1 >$. When *find()* is invoked, it can also delete the duplicated key, i.e $< K_1, V_0 >$ stored in the second sub-table.

III. Detailed Algorithmic Description

We are now presenting the detailed description and pseudo-code for the functions of our cuckoo hash table. The pseudo-code follows the C/C++ language conventions.

A. Search operation

Searching for a key in our lock-free cuckoo hash table includes querying for the existence of the key in two sub-tables, as in Program 2. A key is available if it is found in one of them. A search operation starts with the first query round by reading from the possible slots in the primary sub-table table[0] (lines 24-26), and then in the secondary sub-table table[1] (lines 27-29). If the key is found in one of them, the value mapped to key is returned.

As mentioned in section II, the above query can miss the searched key which happens to be a "moving key". The key present in table[1] is relocated to table[0], meanwhile *search* reads from table[0] and then table[1]. To avoid such missing, *search* performs the second round query (lines 32-34).

Program 1 Data structure and support functions

```
1  HashEntry:
2    word key,
3    word value;

5  CountPtr:
6    <HashEntry*,int> <entry,counter>

8  int hash1(key)
9  int hash2(key)
10 bool checkCounter(int ts1, int ts2, int ts1x, int ts2x)
11   /*check the counter values to see if 2 relocations may
          have taken place*/

13 class CuckooHashTable
14   EntryType *table[2][]      //2 sub-tables
15   word find(word key, CountPtr &<e1,ts1>, CountPtr &<e2,
          ts2>)
16   word search(word key)
17   insert(word key, word value)
18   remove(word key)
19   relocate(int which, int index)
20   help_relocate(int which, int index, bool initiator)
21   del_dup(idx1, <e1,ts1>, idx2 , <e2,ts2>)
```

Program 2 *word* ***search*** (*word* key)

```
22 //h1=hash1(key) and h2=hash2(key)
23 while (true)
24   <e1,ts1> ← table[0][h1] //read the element \& counter

25   if (e1≠NULL ∧ e1→key = key)
26     return e1→value
27   <e2,ts2> ← table[1][h2]
28   if (e2≠NULL ∧ e2→key = key)
29     return e2→value

31   //second round query
32   <e1x,ts1x> ← table[0][h1]
33   ...
34   <e2x,ts2x> ← table[1][h2]
35   ...

37   if (checkCounter(ts1, ts2, ts1x, ts2x))
38     continue
39   else
40     return NIL
```

Program 3 *word* ***find***(*word* key, CountPtr& <e1,ts1>, CountPtr& <e2,ts2>)

```
41 //h1=hash1(key) and h2=hash2(key)
42 word result; int counter
43
44 while (true)
45   <e1,ts1> ← table[0][h1]
46   if (e1 ≠ NULL)
47     if (e1 is marked)
48       help_relocate(0, h1, false)
49       continue
50     if (e1→key = key)
51       result ← FIRST
52
53   <e2,ts2> ← table[1][h2]
54   if (e2 ≠ NULL)
55     if (e2 is marked)
56       help_relocate(1, h2, false)
57       continue;
58     if (e2→key = key)
59       if (result = FIRST)
60         del_dup(h1,<e1, ts1>,h2,<e2, ts2>)
61       else
62         result ← SECOND
63
64   if (result=FIRST ∨ result=SECOND)
65      return result
66   /*second round query*/
67   <e1,ts1x> ← table[0][h1]
68   ...
69   <e2,ts2x> ← table[1][h2]
70   ...
71
72   if (checkCounter(ts1, ts2, ts1x, ts2x))
73     continue
74   else return NIL
```

This two-round query, however, still can miss an existing key if the key is relocated back and forth between table[1] and table[0] repetitively and alternatively when *search* reads from each sub-table. The possibility of such continuously relocation of a key is very rare but can not be ruled out. To deal with that, we employ a technique based on Lamport's logical clocks [15]. The idea of this technique is to attach a counter to each slot of the hash table to record the number of relocations happening at that slot. Similar to a logical clock whose value changes when a local event happens or when a message is received, the value of the counter is changed on the event of relocations. The counter is initialized to 0. When an element stored in a slot is relocated, the slot's counter is incremented. When a slot serves as the destination of a relocation, its counter is updated with the maximum of its current counter value and the source's counter value, plus 1. The counter value of a slot remains even when the element stored in that slot is deleted or relocated to the other sub-table. For example, consider a key associated with counter value t is stored at table[1][h2]. When key is relocated to table[0][h1] which has counter t_1, the new counter value of table[0][h1] is $max(t, t_1) + 1$ and that of table[1][h2] is incremented to $t + 1$.

By examining the above counters after the second round query, a search can detect if it might have missed an existing key. Such missing happens if: (i) Before the execution of line 24, the key is stored in the secondary table at table[1][h2], then (ii) the key is relocated from table[1] to table[0] before the second read at line 27, then (iii) relocated back to table[1] before the next read at line 32, and finally (iv) relocated again back to table[0] before line 34 . If it is so, the counter value read at line 32 should be at least two units higher than the one read at line 24. Similar condition is applied for the counter values read at line 34 and line 27. In addition, the counter value read at line 34 is at least 3 units higher than the one read at line 24 because the counter increases its value like a logical clock when a relocation happens. If these conditions are satisfied, i.e *checkCounter* returns $true$, the two-round query probably misses an item because of alternative relocations, so the search restarts.

In practice, as each slot in the hash table is a pointer to a table element, we can use the unused bits of pointer values on x86_64 to store the counter value. Currently pointers on x86_64 use only 48 lower bits of the available 64 bits. We can use the 16 highest bits of the 64-bit pointer to store the

Program 4 ***insert***(*word* key, *word* value)

```
75 //h1=hash1(key); h2=hash2(key)
76 HashEntry *newNode(key,value)
77 CountPtr *<ent1,ts1>, *<ent2,ts2>
78 start_insert:
79   int result ← find(key, <ent1,ts1>, <ent2,ts2>)
80   if (result=FIRST ∨ result=SECOND)
81     Update the current entry with new value
82     return

84   if (ent1=NULL)
85     if (¬CAS(&table[0][h1],<ent1,ts1>,<newNode,ts1>))
86       goto start_insert
87     return
88   if (ent2=NULL)
89     if (¬CAS(&table[1][h2],<ent2,ts2>,<newNode,ts2>))
90       goto start_insert
91     return

93   result ← relocate(0, h1)
94   if (result=true) goto start_insert
95   else
96     //rehash()
```

counter value, an approach which has been used in literature [16]. This approach is efficient as the pointer to an element and its counter can be loaded in one read operation. The disadvantage of this technique is that the counter which has been increased by $2^{16} + k$ can be misinterpreted to be increased by just k, where k is any counter value. However, such increment of $2^{16} + k$ can only be made by many thousands of relocation operations happening at the same slot. Moreover, it must have happened in a very short period of time of a search operation to cause such misinterpretation. With a good choice of hash functions, the possibility that such misinterpretation happens is practically impossible. Therefore, 16 bits are sufficient to store the counter value.

B. Find operation

Program 3 shows the pseudo code of the *find* operation, which functions similar to the *search*. The *find* takes an argument key and answers if, and in which sub-table, the key exists. In addition, it also reports the current values (and their associated counter values) stored at the two possible positions of key. The logic flow of the *find* is similar to that of the *search*, in the sense that it also uses a two round query. However, it has 3 main differences compared to the *search*. First, if it reads an entry who LSB is marked, indicating an on-going relocation operation, it helps the operation (lines 47-48, and 55-56). Secondly, it examines both sub-tables instead of returning immediately when key is found. This is to discover if two instances of the key exist in two sub-tables. When the same key is found on both sub-tables, the one in the secondary sub-table table[1] is deleted (line 60), as described in Section II. Finally, *find* returns also current items which are stored at two possible slots where key should be hashed to. This information is used by the invokers, i.e *insert* or *delete* operations, as described in next subsections.

Program 5 ***remove***(*word* key)

```
97  //h1=hash1(key); h2=hash2(key)
98  CountPtr *<ent1,ts1>, *<ent2,ts2>
99  while (true)
100   ret ← find(key,<ent1,ts1>,<ent2,ts2>)
101   if (ret = NULL) return
102   if (ret = FIRST)
103     if (CAS(&table[0][h1],<ent1,ts1>,<NULL,ts1>))
104       return
105   else if (ret = SECOND)
106     if (table[0][h1]≠<ent1,ts1>)
107       continue
108     if (CAS(&table[1][h2],<ent2,ts2>,<NULL,ts2>))
109       return
```

C. Insert operation

The insertion of a key, Program 4, works as follows. First, it invokes the *find* at line 79 to examine the state of the key: if it exists in the sub-tables and what are the current entries stored at the slots where the key can be hashed to. If the key already exists, the current value associated with it is updated with the new value and the *insert* returns (line 82). Otherwise, the *insert* operation proceeds to store the new key. If one of the two slots is empty (lines 84 and 88), the new entry is inserted with a CAS. If both slots are occupied by other keys, relocation process is triggered at line 93 to create an empty slot for the new key. The relocation operation is described in detail in Section III-E. If the relocation succeeds to create an empty slot for the new key, the insertion retries. Otherwise, which means the length of the relocation chain exceeds the THRESHOLD, the insertion fails. In this case, typical approaches in the literature of cuckoo hashing such as a rehash with two new hash functions or an extension of the size of the table can be used.

D. Remove operation

The *remove* operation also starts by invoking *find* at line 100. If the key is found, it is removed by a CAS, either at line 103 or 108.

E. Relocation operation

When both slots which can accommodate a newly inserted key are occupied by existing keys, one of them is relocated to make space for the new key. This can trigger a sequence of relocations as the other slot might be occupied too. The *relocate* method presented in Program 6 performs such a relocation process. As mentioned earlier, we use a relocation strategy which can retain the presence of a relocated key in the table without the need for expensive atomicity of the whole relocation process. First, the cuckoo path is discovered, lines 113-135. Then, the empty slot is moved backwards to the beginning of the path, where the new key is to be inserted, lines 137-154.

The path discovery starts from a slot index of one of the sub-tables identified by which and runs at most THRESHOLD steps along the path. If table[which][index] is an empty slot, the discovery finishes (line 134). Otherwise, i.e the slot is

Program 6 ***int relocate***(*int* which, *int* index)

```
110 int route[THRESHOLD] //storing cuckoo path
111 int start_level=0, tbl=which, idx=index

113 path_discovery:
114   bool found ← false
115   int depth ← start_level
116   do {
117     <e1,ts1> ← table[tbl][idx];
118     while (e1 is marked)
119       help_relocate(tbl, idx, false)
120       <e1,_> ← table[tbl][idx]
121     if (<pre,tsp>=<e1,ts1> ∨ pre→key=e1→key)
122       if (tbl = 0)
123         del_dup(idx,<e1,ts1>,pre_idx,<pre,tsp>)
124       else
125         del_dup(pre_idx,<pre,tsp>,idx,<e1,ts1>)
126     if (e1 ≠ NULL)
127       route[depth] = idx
128       key ← e1→key;
129       <pre,tsp> ← <e1,ts1>
130       pre_idx ← idx
131       tbl ← 1 - tbl
132       idx ← tbl = 0 ? hash1(key) : hash2(key)
133     else
134       found ← true
135   } while (!found ∧ ++depth<THRESHOLD)

137 if (found)
138   tbl ← 1 - tbl;
139   for (i ← depth-1; i>=0; --i, tbl ← 1-tbl)
140     idx ← route[i].index
141     <e1,ts1> ← table[tbl][idx]
142     if (e1 is marked)
143       help_relocate(tbl, idx, false)
144       <e1,ts1> ← table[tbl][idx]
145     if (e1 = NULL)
146       continue
147     dest_idx ← tbl=0?hash2(e1→key):hash1(e1→key)
148     <e2,ts2> ← table[1-tbl][dest_idx]
149     if (e2 ≠ NULL)
150       start_level ← i+1
151       idx ← dest_idx
152       tbl ← 1 - tbl
153       goto path_discovery
154     help_relocate(tbl, idx, false)
155 return found
```

occupied by a key (line 126), the key should be relocated to its other slot in the other sub-table. The discovery then continues with the other slot of key. If this slot is empty, the discovery finishes. Otherwise, the discovery continues similarly as before. Each element along the path is identified by a sub-table and an index on that sub-table. Along the path, the sub-tables that elements belong to alternatively change between the primary and secondary ones. Therefore, the data of the path which need to be stored are the indexes of the elements along the path and the sub-table of the last element.

Program 7 Help relocation and delete duplication operations

```
156 void help_relocate(int which, int index, bool initiator)
157   while (true)
158     <src,ts1> ← table[which][index]
159     while (initiator && src is not marked)
160       if (src = NULL) return
161       CAS(&table[which][index)],<src,ts1>,<src|1,ts1>
162       <src,ts1> ← table[which][index]
163     if (src is not marked) return
164     /*hd=hash(src.key) where hash is hash function used
              for table (1-which)*/
165     <dst,ts2> ← table[1-which][hd])
166     if (dst = NULL)
167       nCnt ← ts1>ts2 ? ts1 + 1 : ts2 + 1
168       if (<src,ts1> ≠ table[which][index])
169         continue
170       if (CAS(&table[1-which][hd],<dst,ts2>,<src,nCnt>))
171         CAS(&table[which][index],<src,ts1>,<NULL,ts1+1>)
172         return
173     //dst is not null
174     if (src = dst)
175       CAS(&table[which][index)],<src,ts1>,<NULL,ts1+1>)
176       return
177     CAS(&table[which][index],<src,ts1>,<src&˜1,ts1+1>)
178     return false;

180 void del_dup(idx1, <e1,ts1>, idx2 , <e2,ts2>)
181   if (<e1,ts1>≠table[0][idx1] ∧
182       <e2,ts2>≠table[1][idx2])
183     return
184   if (e1→key ≠ e2→key)
185     return
186   CAS(&table[1][idx2],<e2,ts2>,<NULL,ts2>)
```

Once the cuckoo path is found, the empty slot is moved backwards along the path by a sequence of “swaps” with the respective preceding slot in the path. Each swap is actually a relocation of the key in the latter slot, a.k.a the source, to the empty slot, a.k.a the destination. Because of the concurrency, the entry stored in the source might have changed. Thus, the relocation operation needs to update the destination and check for its emptiness (lines 148-149), and retry the path discovery if the destination is no longer empty (line 153). If the destination is empty, the relocation is performed in three steps in the *help_relocate* operation presented in Program 7. First, the source entry’s LSB is marked to indicate the relocation intention (line 161). Then, the entry is copied to the destination slot (line 170), which has been made empty. Finally, the source is deleted (line 171). Marking the LSB allows other concurrent threads to help the on-going relocation, for example at lines 119 and 143.

After a slot is marked in *help_relocation*, the destination of the relocation might have been changed and is no longer empty. This can be because either other threads successfully help relocating the marked entry, or a concurrent insertion has inserted a new key to that destination slot. If it is the former case (line 174), the source of the relocation is deleted either by that helping thread (line 171) or by the current thread (line 175). If it is the latter case, the *help_relocation* fails, unmarks the source (line 177) and returns. The relocation process then continues but might need to retry the path discovery in the next loop.

IV. Proof of correctness

In this section, we are going to prove that our cuckoo hash table is linearizable and lock-free. At first, we prove the linearizability under the assumption that a key exists in only one sub-table. Later on, we prove that if there are two instances of a key in two sub-tables, the linearizability is not violated. Then we continue to prove the lock-freedom. We provides proofs related to the *search* operation and the full proofs can be found in the technical report [17]. In the followings, we assume that a key can be stored in two possible positions: table[0][h1] and table[1][h2].

Lemma 1: When *help_relocation*(which, index) is invoked, either it succeeds relocating the element pointed to by table[which][index] to the other table, i.e table (1-which) and unmarks the source slot; or if it fails doing that, the source slot, i.e table[which][index] is unmarked.

Corollary 1: If *help_relocation* succeeds relocating entry in table[which][index] to the other sub-table, the counter values of both source and destination increases by at least 1.

Following Corollary 1, it is easy to see that if there were two relocations which had happened at one slot, the slot's counter would have been increased by at least 2 units.

Lemma 2: The *search* is linearizable.

Proof: The *search* operation returns a value only when one of the keys in e1, e2, e1x, e2x (lines 24, 27, 32 or 34) matches the searched key. In this case, *search* is linearized at the respective line. We now consider the loop of the *search* to see when it returns NIL.

As we discussed in Section III-A, during a search for key, the two round query protocol misses the searched key only when there is a sequence of relocations of key from table[1] to table[0], back to table[1] and again to table[0], as described in subsection III-A. Condition at line 40 can detect if such a scenario may happen by examining the counter values of table[0][h1] and table[1][h2], as described also in subsection III-A. If the condition is satisfied, the *search* restarts. We note that the condition is also satisfied if there are two independent relocations of other keys which are stored in the same slots; or if there is only one relocation at each slot but the relocation increases the counter by two or more units. In these cases, the search might restart unnecessarily but its correctness is not violated.

Therefore, the *search* returns NIL when the key is not found in any of the read entries: e1, e2, e1x, e2x and there is no possibility that the key is relocated which forces the *search* to reexecute the loop. Even though, there are cases that key appears in one of the sub-tables at the time *search* performs a reading from the other sub-table. In such cases, *search* might still return NIL, but we can argue that it is totally correct. We consider, as an example, a key exist in one sub-table as *search* performs reading for the second round query at lines 32 and 34, and show the correct linearization points. Other readings, e.g lines 24 and 27, can be argued in a similar manner. If the key exists in the table when search executes lines 32 and 34, and the *search* returns NIL:

- key must be inserted to table[0] after the reading from that sub-table at line 32. The *search* can be linearized at line 32 where key has not been inserted to the table.
- Or key exists in table[1] before the *search* starts and is deleted before the *search* reads from it (line 34). The *search* can then be linearized to line 34, when key has been deleted.
- Or key exists in table[1] when the *search* reads from table[0] (line 32), is deleted and then re-inserted to table[0] before the *search* reads from table[1] (line 34). In such scenario, neither line 32 or line 34 can be the correct linearization point of the *search*. Because key exists in the table at those points of time, in particular, in the other sub-table than the one *search* reads from. Even though, we notice that there is an interval between when key is deleted from table[1] and when it is inserted to table[0] (if these operations overlap, the re-insertion would have failed), this period of time is inside the duration that *search* executes line 32 to line 34. In this interval, key does not exist anywhere in the table. Therefore, we can always linearize *search* to a point of time in that interval. This satisfies the requirement that the linearization point must be between the time when *search* is invoked and when it responds.

Henceforth, *search* is linearizable. ■

Lemma 3: The *find* operation is linearizable.

Lemma 4: The *remove* operation is linearizable.

Lemma 5: The *insert* operation is linearizable.

Now we consider the scenario where two instances of a key co-exist in the table. When two concurrent insertions try to insert the same key to two sub-tables, they might both succeed and store it in two possible positions of that key. As described in Section II, our hash table allows such physical co-existence and then removes the instance in the second sub-table before any mutating operations can operate on these two positions. The subsequence operations can only see one valid instance, i.e the one in the primary table.

Lemma 6: The correctness of the *search* operation is immune of the concurrent physical existence of key in both sub-tables.

The below propositions can be derived from the pseudo code of *find* and *relocate*.

Proposition 1: If two instances of key co-exist in the table when a *find* on key is invoked, it removes the instance of the key in the second sub-table.

Proposition 2: If two instances of key co-exist in the table when a *relocate* of key happens, it removes the instance of the key in the second sub-table.

Now, we examine the effect of the existence of a duplicated key to the correctness of *insert*/*remove* operations.

Lemma 7: The correctness of a *remove* and *insert* operation of a key in Lemma 4 and 5 holds even when two instances of a key exist concurrently.

Theorem 1: The hash table algorithm is linearizable.

Proof: Each operation of the hash table is linearizable follows Lemmas 2, 4, 5, and Lemmas 6, 7. ■

Now, we are ready to prove that the algorithm is lock-free.

Lemma 8: Either the *help_relocation* operation finishes after a finite number of steps or another operation must have finished in a finite number of steps.

Lemma 9: If a *help_relocation* operation finishes but fails to relocate table[which][index], there must be another operation making progress during that execution of the *help_relocation*.

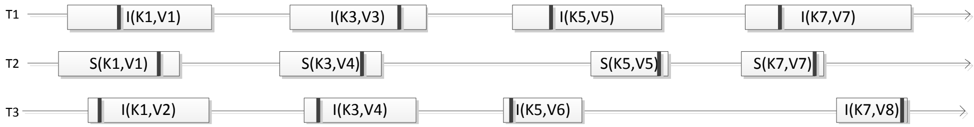

Figure 2: Concurrent inserts can create the existence of two instances of a key in the table

We observe that *help_relocation* can encounter an ABA problem[3] [8], even when we have a proper memory management to handle the hash table's element. This scenario can take place when there are threads executing line 166 to relocate the same table[which][index]. Meanwhile, a new key can be inserted to dst, and then deleted from dst. Some of the threads doing relocation can observe that dst is pointing to the inserted element, which leads to the CAS at line 170 to fail and the source of the relocation to be unmarked at line 177. Meanwhile, other threads doing relocation might perform the CAS at line 170 after the deletion, and therefore succeed to copy table[which][index] to dst. As a result, the key exists in two sub-tables. However, such co-existence caused by the ABA does not hurt the correctness of our algorithm. This is because our algorithm is capable of tolerating such co-existence and can soon remove the one in the second sub-table. Such removal is done by the calling thread performing this *help_relocation*, or any other thread doing *find* or *relocate* which involves the slots storing the duplicated key (at line 60, 123 or 125, as discussed in Section II).

Lemma 10: The *search* operation finishes after a finite number of steps, or another operation must have finished after a finite number of steps.

Proof: The *search* operation has only one while loop which, according to the proof of Lemma 2, repeats only when there are relocations of keys stored in table[0][h1] and table[1][h2]. Such relocations mean progress of the respective operations performing them. This observation holds even when the relocations move key(s) back and forth between two sub-tables. In this case, the *search* might not make progress but the relocation progresses towards the THRESHOLD number of relocation steps. When it reaches the THRESHOLD and returns false, the insertion calling such a relocation fails and proceeds with rehashing or resizing the hash table. ■

Lemma 11: A *find* operation finishes after a finite number of steps, or another operation must have finished after a finite number of steps.

Lemma 12: An *insert* operation finishes after a finite number of steps or another operation must have finished after a finite number of steps.

Lemma 13: A *remove* operation finishes after a finite number of steps or another operation must have finished after a finite number of steps.

Theorem 2: The hash table algorithm is lock-free.

Proof: According to Lemmas 10, 12, 13, our cuckoo hash table always makes progress after a finite number of steps. ■

V. Experimental Evaluation

This section evaluates the performance of our lock-free cuckoo hash table and compares it with current efficient hash tables. We use micro-benchmarks with several concurrent threads performing hash table's operations, a standard evaluation approach taken in the literature.

A. The Experimental Setup

We compare our lock-free cuckoo hashing with:

- A lock-based chained one: that uses a linked-list to store keys hashed to the same bucket. A number of locks, equal to the number of table segments, are used [14].
- Hopscotch hashing: a concurrent version of hopscotch hashing, with each lock for a segment [7]. Thanks to the kindness of the authors, we could obtain the original source code of hopscotch hashing.
- LF Cuckoo: our new lock-free cuckoo hashing.

All the algorithms were implemented in C++ and compiled with the same flags. No customized memory management was used. In all the algorithms, each bucket contains either two pointers to a key and a value, or an entry to a hash element, which contains a key and a value.

The experiments were performed on a platform of two 8-core Xeon E5-2650 at 2GHz with HyperThreading, 64GB DDR3 RAM. In our evaluation, we sampled each test point 5 times and plotted the average. To make sure the tables did not fit into the cache, we used a table-size of 2^{23} slots. Each test used the same set of keys for all the hash tables.

B. Results

Figure 3 presents the throughput result of the hash tables in different distributions of actions. The commonly found distribution is $90s/5i/5r$, i.e 90% *search*, 5% *insert* and 5% *remove*. Other less common distributions were also evaluated. One with more query-only operations: $94s/3i/3r$. Two others with more mutating operations: $80s/10i/10r$ and $60s/20i/20r$. As it is commonly known that original cuckoo hashing works with load factors lower than 49%, we used the load-factor of 40%. The concurrency increased up to 32 threads, the maximum number of concurrent hardware threads supported by the machines.

Our lock-free cuckoo hashing performs consistently better than both the lock-based chained and the hopscotch hashing

[3] ABA problem happens when an operation succeeds because the memory location it read has not changed; but in fact, it has changed its value from A to B and then back to A.

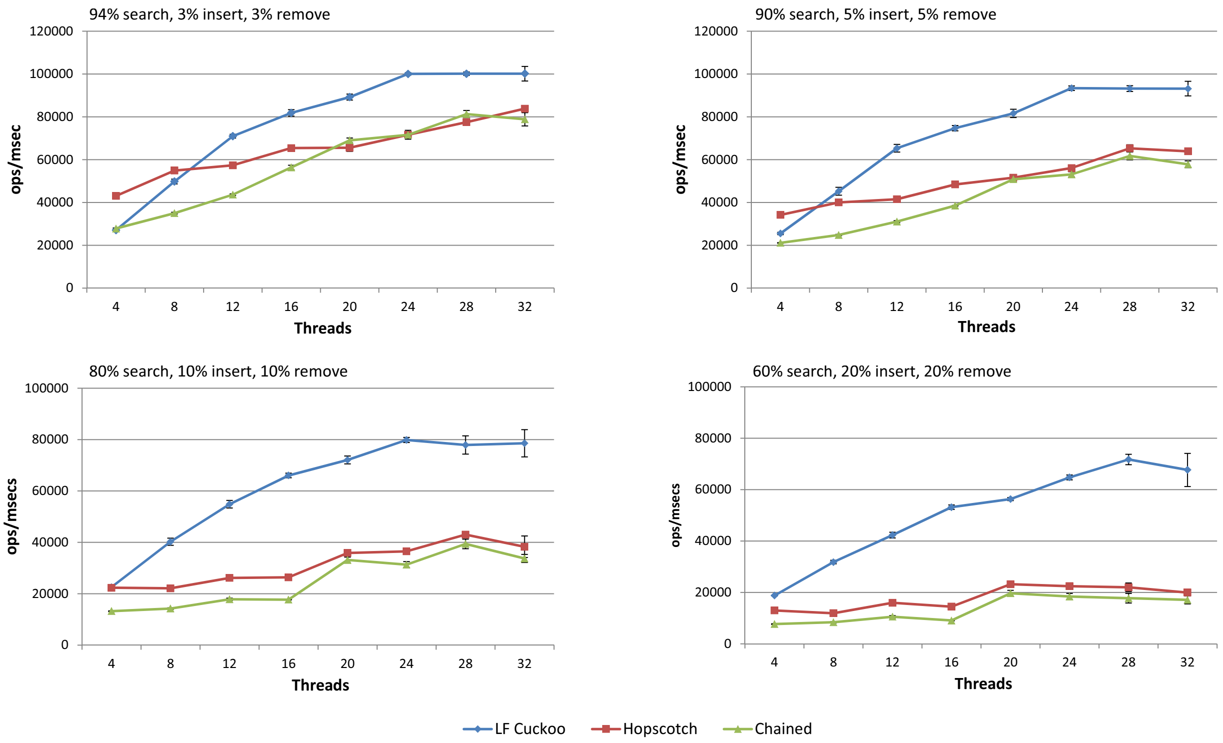

Figure 3: Throughput as a function of concurrency at load factor 40%

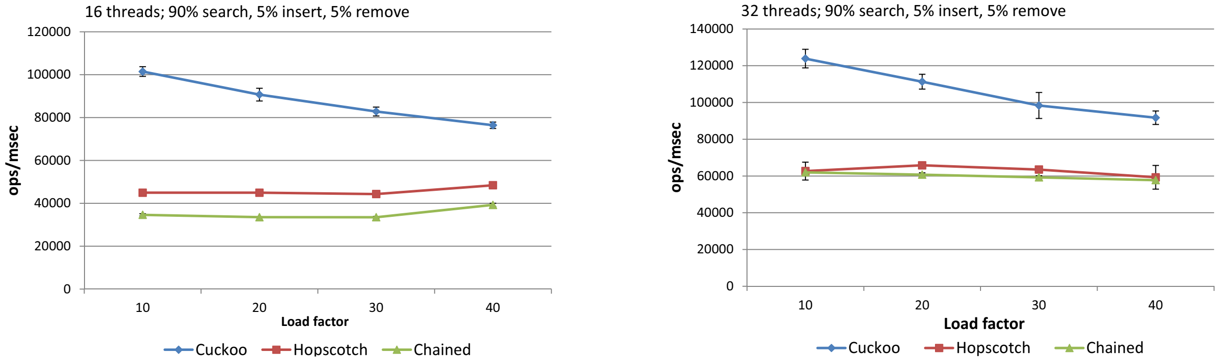

Figure 4: Throughput as a function load factor at 16 and 32 threads

in all the access distribution patterns. This is because positive searches need to examine either one or two table's slots, and negative searches need 3 read operations in most cases. Cases that *search* might need to perform more read operations do happen but not often. This is because the possibility that a relocation of a key happens concurrently as the key is being queried is not high. In addition, our algorithm is designed so that the search operations still make progress concurrently with any other mutating operations. In contrast, the lock-based chained and the hopscotch hashing lock the bucket during insertion or removal.

Our lock-free cuckoo hashing maintains very high throughput in scenarios with more mutating operations, i.e $10i/10r$ or $20i/20r$, respectively. The insert operation in cuckoo hashing might require relocations of existing keys to make space for the new key. The algorithm, however, has been fine designed to allow high concurrency between relocation operations and other operations. Meanwhile, the lock-based chained and the hopscotch hashing degrade quickly when the percentage of mutating operations increases, mainly because of their blocking designs.

Figure 4 presents throughput results as a function of load factor. In cuckoo hashing, higher load factor means more relocations of existing keys during insertion. Therefore, we can observe that the throughput of our lock-free cuckoo hashing decreases when the load factor increases. Nevertheless, our cuckoo hashing algorithm always achieve throughput $1.5-2$ times as much as other algorithms, in both cases of 16 and 32 concurrent threads.

We now analyze the cache behavior of our lock-free

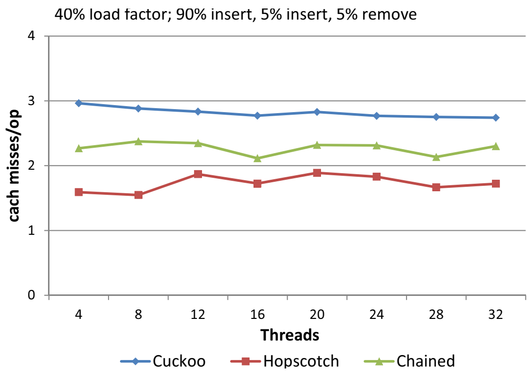

Figure 5: The number of cache-misses per operation.

cuckoo hashing. A positive search operation usually requires reading 1 or 2 references and a negative one often requires reading 3 references in the two-round query protocol, if no concurrent relocation happens at the read slots. Otherwise, search operations might need to perform more read operations, which might cause more cache misses. A removal of a key often needs one additional CAS compared to a search operation to delete the found element from the tables. Insertion operations are more complicated. If an insertion does not trigger any relocation, its behavior is similar to the deletion operation. Otherwise, it can cause more cache misses. We have recorded that the number of relocations is approximately 2% of the total performed operations, in the distribution of $90s/5i/5r$. Therefore, we expect a higher number of cache misses in our cuckoo hashing compared to other hash tables. A measurement of number of cache misses of our lock-free cuckoo hashing is presented in Figure 5. Our lock-free cuckoo table triggers about 3 cache misses per operation, a bit higher than hopscotch hashing and lock-based chained hashing. Regardless of a slightly higher number of cache misses, our lock-free cuckoo hashing has maintained a good performance over the other algorithms, thanks to the fine designed mechanism to handle concurrency.

VI. Conclusion

We have presented a lock-free cuckoo hashing algorithm which, to the best of our knowledge, is the first lock-free cuckoo hashing in the literature. Our algorithm uses atomic primitives which are widely available in modern computer systems. We have performed experiments that compares our algorithm with the efficient parallel hashing algorithms from the literature, in particular hopscotch hashing and optimized lock-based chained hashing. The experiments show that our implementation is highly scalable and outperform the other algorithms in all the access pattern scenarios.

Acknowledgment

The authors would like to acknowledge the support of the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183.

References

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

[2] M. Zukowski, S. Héman, and P. Boncz, "Architecture-conscious hashing," in *Proceedings of the 2nd International Workshop on Data Management on New Hardware*. ACM, 2006.

[3] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," in *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, ser. LNCS, vol. 2607. Springer Berlin Heidelberg, 2003, pp. 271–282.

[4] K. Ross, "Efficient hash probes on modern processors," in *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE)*, 2007, pp. 1297–1301.

[5] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM J. Comput.*, vol. 39, no. 4, pp. 1543–1561, Dec. 2009.

[6] D. Lea, "Hash table util.concurrent.concurrenthashmap," http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/, 2003.

[7] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing," in *The Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, ser. LNCS, vol. 5218. Springer Berlin Heidelberg, 2008, pp. 350–364.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[9] P. Tsigas and Y. Zhang, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 29, no. 1, pp. 320–321, Jun. 2001.

[10] D. Cederman, A. Gidenstam, P. H. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas, "Lock-free concurrent data structures," in *Programming Multi-Core and Many-Core Computing Systems*, S. Pllana and F. Xhafa, Eds. Wiley-Blackwell, 2014.

[11] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 2002, pp. 73–82.

[12] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379–405, May 2006.

[13] C. Click, "A lock-free wait-free hash table," http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf, accessed: 2013-11-14.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[16] M. Brunink, M. Susskraut, and C. Fetzer, "Boundless memory allocations for memory safety and high availability," in *Proceedings of the 41st International Conference on Dependable Systems Networks (DSN)*, 2011, pp. 13–24.

[17] N. Nguyen and P. Tsigas, "Lock-free cuckoo hashing," Chalmers University of Technology, Department of Computer Science and Engineering, Tech. Rep. 2014:03, January 2014.