A Non-blocking Hopscotch Hashing algorithm

Ziaul Choudhury, Suresh Purini
International Institute of Information Technology, Hyderabad, India
Email: {ziaul.c@research. suresh.purini} @iiit.ac.in

***Abstract*—This work presents a lock-free hash table based on the hopscotch probing and displacement technique. The hopscotch algorithm is an open address hashing method combining ideas from chaining, cuckoo hashing and linear probing. Our proposed hash table has minimum storage overhead and places the key-value pairs into a single memory block. Also, the use of less atomic operations and more structured memory accesses, lowers the overhead of synchronization and cache misses respectively.**

In a series of micro-benchmarks with 8 million unique key-value pairs on a 16 core `Intel` machine, our hash table achieves 1.5x higher throughput compared to a recently proposed lock-free version of the `Cuckoo` hash table and about 2x higher throughput compared to a popular lock-free resizeable hash table. We measured the throughput in million queries processed per second (MQPS). Also, at higher load factors (0.6 and above) our hash table performs consistently well with better throughput and lower failure rate compared to the existing alternatives.

I. Introduction

A hash table is a key-value store supporting constant time storage and retrieval operations with high probability. There exists two broad classes of hashing algorithms in general namely open addressing and closed addressing. Linear probing, double hashing, chaining and cuckoo hashing are few popular representative hash table designing techniques that can be categorized under these two classes [1], [6]. With the recent uprise of multi-core computers and popularizing of parallel computing, efforts have been made towards designing a concurrent thread safe hash table. A high throughput concurrent hash table lies at the heart of data intensive applications, e.g databases, bioinformatics, operating on a multi-core processor with hundreds of active threads running queries in parallel. With interleaved queries, a correct implementation of the concurrent hash table guarantees a serializeable dynamic set interface.

There exists many versions of a concurrent hash table in literature. Lea's hash table from the `Java` Concurrency Package [5] is a closed address hash table based on chaining and implemented using locks. Lock-free algorithms designed using the underlying atomic primitives have proven to achieve better scalability and throughput compared to lock based algorithms. Micheal [7] presented an efficient lock-free hash table with separated chaining using linked lists. Shalev and Shavit [10] designed another high performance lock-free closed address resizeable hash table using a recursive split-ordering technique (a specific ordering of elements in a linked list where they can be repeatedly "split using atomic exchange operations). Nguyen presented a lock-free cuckoo hash table [8]. The algorithm successfully supported concurrent operations using a two round query protocol enhanced with a logical clock technique and by breaking the key relocation path evident in the cuckoo hash technique into several single relocations.

Altough, the aforementioned hash tables are promising, they do have a few limitations. For example, the relocation path in cuckoo hashing can be long in the presence of too many concurrent operations. The idea of hopscotch hashing was recently introduced by Herlihy and Shavit [3]. Hopscotch hashing is an open addressing based hashing algorithm that has the flavors of chaining, cuckoo hashing [9], and linear probing, all put together, yet avoiding the limitations and overheads of these former approaches. The present concurrent version of hopscotch hashing is implemented using locks. Every hash table slot has an associated lock that synchronizes the concurrent accesses. Under clustering and collision effects, mutating operations can trigger a chain of lock acquisitions leading to serialization, long delays, priority inversion and other such typical drawbacks associated with lock based designs [2].

In this paper, we propose the first lock-free hopscotch algorithm using the single word compare and swap atomic primitive and report a higher throughput compared to present state of art lock-free hash tables. The specific contributions in this direction are as follows.

- We demonstrate a novel way of designing block allocated lock-free hash tables with the bare minimum space[1]. Block allocated data structures, i.e. data structures that live inside a single memory block, relieve the implementation from overhead introduced by the actions of the garbage collector.
- Our algorithm has low synchronization overhead and has sequential memory access patterns leading to better cache utilization.
- We prove the correctness of our lock-free implementation with a linearization argument and compare its performance with most effective prior know lock-free hash tables.

[1]Here by minimum we mean the space equivalent to storing a sequential hash table storing just the key-value pairs and no extra memory being used towards making it concurrent.

The layout of the rest of the paper is as follows. In Section 2 we give a brief background on hopscotch hashing. Sections 3 through 5 describes our algorithm in details. Followed by experimentation and conclusion in section 6 and 7 respectively.

II. Hopscotch Hashing

The main idea behind hopscotch hashing (figure 1) is that each hash table slot has a neighborhood of size H. A key is either present in its designated slot as per the hash function or in the adjacent $H-1$ slots. The parameter H is a feature of the hopscotch algorithm called the hopscotch range. A slot along with the adjacent $H-1$ slots make a virtual bucket which also means that at any slot, multiple neighborhoods are overlapping (H to be exact). Each slot maintains a bitmap of size H called `HopInfo` that maps the information as to which of the adjacent H slots have keys relating to the respective virtual bucket.

The present concurrent version of the hopscotch algorithm uses a fine grained locking mechanism where each virtual bucket is mapped to a lock using the hash table slots. This lock controls accesses and modifications to all the entries, reflected in the `HopInfo` bitmap, mapped to that slot (virtual bucket). The `contains()` method is wait-free and relies on per slot time stamps to order the read/writes to that particular slot. The time stamp data ensures that the `contains()` method views the most recent version of the `HopInfo` bitmap, Each time the `contains()` method detects an update in the bitmap via the change in the slot time stamp, it re-executes itself, which is being modified by concurrent mutating operations. It is extremely cache friendly as it requires at most H consecutive look ups in contiguous memory locations starting from the initial hashing slot.

Insertions proceed by locking and trying to place the key at the base slot B to which the key is initially hashed. On failure, an empty slot E is discovered using probing, $E >= B$. If $|B-E| > H$, E is repeatedly swapped (each time at distance of $H-1$ from the previous slot) with other slots, locking each participating slot in the process, and relocated to the neighborhood of B. Before releasing the locks, the `HopInfo` bitmap of each locked slot is updated to reflect the new locations of the relocated entries. During relocation, it is always ensured that the hopscotch guarantee is not broken for any key, i.e. no key is moved out of its own virtual bucket. Deletions acquire the single lock associated with B to remove the key and modifies the bitmap before releasing the lock.

Designing a lock-free version of the hopscotch algorithm has two major challenges. Since a bitmap cannot be operated atomically, the keys mapped to each virtual bucket needs to be mapped correctly in the presence of concurrent operations. Secondly, we need to make sure that a relocation operation does not render a key untraceable. To address the bitmap issue, we resort to sequential scanning of a virtual bucket instead of hopping inside the bucket. Virtual buckets are traced using array indices. To ensure safe relocations, we use atomic markers embedded inside hash table slots and correctly serialize the concurrent relocate with other hash table operations.

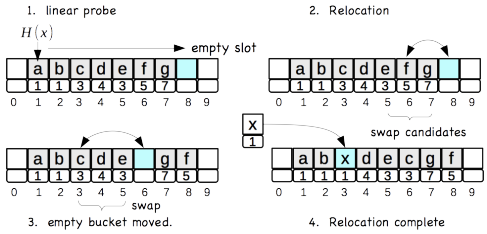

Figure 1: The figure demonstrates the hopscotch algorithm. To insert a key *x*, an empty slot (slot 8) is found using linear probing. If the distance between the empty slot and the initial hashing location is greater than the hopscotch range (4 in this case), then the empty slot is brought in the hopscotch range using a sequence of swaps. Each time the empty slot e is exchanged with a slot b towards the left such that after relocation b is still within its hopscotch range. The index of the base slot is shown with the small rectangle below each array entry. Finally, *x* is placed in third array slot, with virtual bucket number 1.

III. Overview of the Proposed Hash Table

Our lock-free hopscotch hash table is an array of 64 bit slots ($B[0...n]$). Each slot contains a 32 bit key and a 32 bit value [2]. The 2 most significant bits from the value part is used to store state information for the respective slot. Every slot can be in any of these states (determined by the value of the status bits) listed below. A slot taking part in an insert or delete operation undergoes through these states.

- `INIT('00')` – This is the default state of a slot that permits the contents of the slot to be looked up, removed or relocated.
- `REL('01')` – This state marks a slot for relocation. Once marked a slot can only be relocated after the current relocation is over. .
- `INSERTING('11')` – A slot in this state contains the most recent key, that is currently undergoing insertion by a thread. Note that the key in this state cannot be discovered by a look up operation. This state is used for duplicate key elimination and serializing concurrent relocation and mutating operations.
- `RES('10')` – This state is used when the hash table is going through a rebuilding phase. Before the entries in the hash table are re-hashed to a new location, each slot is brought to this state. This is done so that the threads still operating with the older version of the hash table can know of the resizing, thereby abandoning their current operation and helping in the resizing effort instead.

A hash function $H_f()$ (we use a 64 bit version of the Murmur hash function). is used to hash the key to a hash table slot. While a key-value insertion is in progress, a certain number of bits from the value is reserved to store a unique

[2]For variable sized value, a 32 bit pointer pointing to the value can be stored in the value section of each slot.

thread identifier for the thread performing the insertion. This is done to avoid the occurrence of duplicate key-value pairs. Once the key is inserted and made visible to future queries on the hash map, the thread identifier bits are cleared and the value is restored to its original form. The final placement of the key and the restoration of the value is done in one atomic operation. The mutating operations for our hash table are engineered to meet the hopscotch invariant, i.e. for any key x present in the hash table, it either belongs to the slot $\mathtt{B}[\mathtt{H_f}(\mathtt{x})]$ or any one of the slots $\mathtt{B}[\mathtt{H_f}(\mathtt{x})+1]$ through $\mathtt{B}[\mathtt{H_f}(\mathtt{x})+\mathtt{H}-1]$, H is the hopscotch range.

The insert and delete operations in our hash table are lock-free while the look up is wait-free. In the original hopscotch algorithm an empty slot is moved to a proper index in the array before the key is finally inserted. We follow an "optimistic" design and place the key based on the first probe sequence inside an empty slot, put it to a INSERTING state, and then relocate the slot as required based on the length of the probe sequence as was described before. The state of the slot is upgraded to INIT once it is placed inside its respective virtual bucket. While relocating a key from a source slot to a destination slot, the relocated key is not deleted from the source until it is copied to its destination. This dual positioning of the key for a transient period of time guarantees that a concurrent look up operation does not miss an existing key which is getting relocated concurrently. The remove operation scans H consecutive locations of the hash table for the key. During this scan, it also helps towards the correct execution of a concurrent relocation happening in that range.

IV. Detailed Description

In this section we describe our lock-free algorithm in details.

A. contains

A contains(x) looks for the key x inside its virtual bucket.The method scans slots, for the key, within the range $\mathtt{B}[\mathtt{H_f}(\mathtt{x})]$ through $\mathtt{B}[\mathtt{H_f}(\mathtt{x})+\mathtt{H}-1]$ that are either in the INIT or the REL state. It is wait-free as it performs a constant number atomic comparisons within the designated slot range without waiting on any condition. Unlike the locked version, per slot time stamps are no longer required as the ordering of a concurrent look up and mutating operation happens through the relative ordering of the atomic update and read operations respectively.

B. add

The add(x) (algorithm 1) method places a key x at a slot $\mathtt{B}[\mathtt{m}]$ such that $|\mathtt{H_f}(\mathtt{x})-\mathtt{m}| \leq \mathtt{H}$. Firstly, a new 64 bit entry E containing the new key-value pair is created and put to a INSERTING state by stealing the two most significant bits from the value. Also, the unique thread identifier corresponding to the thread performing the insertion

Algorithm 1 Inserting a key/value pair [k,v] in the hash map.

```
1:  procedure ADD(K,V)
2:      Index ← H_f(k)
3:      base ← Index
4:      E = makeEntry(k,mytid) & E.state = INSERTING
5:      if contains([k,v]) then return
6:      end if
7:      do   ▷ The isGrowing function checks for the resizing state of hash table.
8:          if isGrowing() then
9:              resize() and retry
10:         end if
11:         while Slots[Index] ≠ 0 do
12:             Index ← Index + 1
13:         end while
14:     while !Slots[i].CAS(0,E)   ▷ Linear Probe to find an empty slot to place the new entry E.
15:     while Index − base > H do ▷ Length of the probe sequence is greater than the hopscotch range H.
16:         Index ← Relocate(Index,E)   ▷ Keep relocating E until it is inside its virtual bucket.
17:     end while
18:     if Validate(base,x) = True then   ▷ Check for other threads inserting the same key in the virtual bucket.
19:         do
20:             if isGrowing() then
21:                 resize() and retry
22:             end if
23:             ExpEntry ← Slots[Index]
24:             tid ← getID(ExpEntry)
25:             if tid ≠ myTid then   ▷ Another thread with a lower thread ID is inserting this key into the virtual bucket go to 29
26:             end if
27:         while !Slots[Index].CAS(ExpEntry,[k,v])   ▷ The value is restored at this point and state of the slot is reset to INIT marking the end of a successful insertion.
28:     end if
29:     do
30:         if isGrowing() then
31:             resize() and retry
32:         end if
33:         ExpEntry ← Slots[i]
34:     while !Slots[Index].CAS(ExpEntry,0)   ▷ Another thread has taken the responsibility of inserting this key/value pair so clear the slot that this thread was using for insertion.
35: end procedure
```

is embedded in the value part. Now, E is placed in an empty slot using linear probing (probe sequence starts at $H_f(x)$). After the successful placement of E, the hopscotch invariant is checked. If the probe sequence length is greater than H, E is brought into the virtual bucket of x, starting at $H_f(x)$, through repeated invocations of the relocate method.

A single invocation of the relocate method (algorithm 3) brings x closer to its virtual bucket by swapping E with a slot on its left, returning the new location of E. The $H-1$ slots preceding E are scanned for a swap candidate. Once a slot B[j] is found such that $i - H_f(B[j].key) + 1 <= H$, i is the location of E, and B[j] is in the INIT state, it is marked for relocation by changing its state to REL (line 15-17 in the algorithm). This marking prevents the candidate slot from moving off its place due to a concurrent relocate. After the marking, the physical relocation happens. The contents of B[j] is copied to B[i] using one compare and swap operation (lines 18-24). The state of B[i] is now INIT which was previously INSERTING when it contained E. At this point, the swapped key is present in two slots, ensuring that a concurrent look up does not miss the key while it is is being relocated. Now E, without altering its state, is copied to B[j] (lines 25-30). This unchanging of state makes sure that E does not again shift right due to a concurrent relocate. The state of E is reset to INIT only when it moves inside the virtual bucket of x. Further calls to relocate are done based on its return value. If the relocate method fails to find a candidate slot, it triggers a resize and returns an error flag.

Algorithm 2 The validation step within the add method

```
1:  procedure VALIDATE( B, X )
2:      for E in Slots[i], b <= i < b + H and [key]E = x do
3:          if [state]E = INIT then   ▷ The key/value pair is in the map already. return False
4:          end if
5:          tid ← getID(v)            ▷ The ID of the thread inserting the key/value in the map.
6:          if tid < myTid then return False
7:          end if
8:          if tid > myTid & tid ≠ MARKER then
9:              Entry ← Mark(E)
10:             if !Slots[i].CAS(E,Entry) then
11:                 if [key]E = k & [state]E = INIT then   ▷ Re-read the contents of E to verify if the key has been removed or inserted by another thread. return False
12:                 end if
13:             end if
14:         end if
15:     end for
16:     return True
17: end procedure
```

Validation: Assume a scenario, where two threads via two valid probe sequences place the same key x at two different slots and puts them in the INSERTING state. Eventually when both the threads try to place x inside its virtual bucket, they both may end up placing x at different slots of the same virtual bucket resulting in a duplication.

We device a "leader election" algorithm using the thread identifiers, stored in the slots, to resolve this. After the relocation phase ends and the new entry (the slot containing it is still in the INSERTING state) moves inside the respective virtual bucket. The thread t executes the Validate method (algorithm 2) scanning for the thread identifiers and keys stored in each slot of the virtual bucket. On finding a slot containing x and having a lesser thread identifier[3] than t, the method returns false (line 6).

If it could not find such a slot, the method marks the slots having thread identifiers greater than t (lines 8 -10), by overwriting the thread identifier with a specific marked value. This is done atomically with a single CAS (line 10). For our implementation we choose the marker value to be greater than all the thread identifier values. The failure of the CAS at line 10 can mean two things. Either a remove(x) is successful or the larger thread has already completed its insertion of x. The latter case if true if the slot under consideration still contains x and is in the INIT state (check in line 11). In which case the validate returns false.

A natural question to ask here is how do we differentiate between between a value and a thread identifier. A thread might mistake a value as a thread identifier and clear the value instead compromising the correctness of the hash map. To resolve this, a check is performed inside the validate method (line 3). If the value part contains a value instead of a thread identifier, than the slot containing x is already in the INIT state and the x is a part of the hash map. In other words a previous thread has already completed the insertion of x, in which case the validate method returns false.

Now the add method, based on the return value of the validate method, either changes the state of the new slot to INIT clearing the thread identifier (line 27 of the add method) and restoring the original value, or completely erases its contents (line 34 of the add method), including the state information with a single compare and set operation. The check on line 25 of the add method, is to signal the larger threads that their slots have been marked by a thread with a lower identifier value, thus they can proceed for erasing the contents of the respective slot that they were keeping hostage to complete their insertion of x. In simpler words, in a scenario of multiple concurrent insertions of the same key by different threads, our algorithm elects the thread with the lowest identifier value to complete the insertion.

[3]The thread identifiers for the slots in a non INSERTING state is taken as 0.

Algorithm 3 Relocating an Entry E to its virtual Bucket.

```
1: procedure RELOCATE(LOC,E)
2:     Replace ← loc − H & Ind ← Replace ▷ looking for the swap candidate.
3:     do
4:         BaseLoc ← getSlot(lnd)
5:         HopDist ← Loc − BaseLoc
6:         while HopDist > H & Replace < Loc do
7:             Replace ← Replace + 1
8:             BaseLoc ← getSlot(Replace)
9:             HopDist ← Loc − BaseLoc
10:        end while
11:        if HopDist > H||Replace >= loc then
12:            resize() and retry
13:        end if
14:        Exp = Final ← Slots[Replace]
15:        Exp.state = INIT & Final.state = REL
16:        Ind ← Replace + 1        ▷ Attempt to mark the swap candidate for relocation.
17:    while !Slots[Replace].CAS(Exp,Final)
           ▷ The swap candidate is relocated within its virtual bucket and its state is reset to INIT.
18:    do
19:        if isGrowing() then
20:            resize() and retry
21:        end if
22:        Dest ← Slots[loc] & Dest.state = [11]
23:        Final ← Slots[Replace] & Final.state = [00]
24:    while !Slots[loc].CAS(Dest,Final)
       ▷ The new entry E is relocated to the location of the swap candidate.
25:    do
26:        if isGrowing() then
27:            resize() and retry
28:        end if
29:        Exp ← Slots[Replace] & Exp.state = REL
30:    while !Slots[Replace].CAS(Exp,E)
31:    return Replace
32: end procedure
```

C. remove

A remove(x) (see algorithm 3) method looks for the key *x* inside its virtual bucket. Once a slot containing *x* and either in the INIT or the REL state is found, the contents of the slot are cleared except the state information (status bits) with a single compare and swap operation. The state information of the slot needs to be preserved as it may be taking part in a concurrent relocate. Changing its state would alter the correct execution flow of the relocation. After this the method scans through rest of the slots in the bucket helping towards the correct execution of a concurrent relocate, happening within the same slot range, see Figure 2. Assume a relocation where a slot B[i] has been marked for relocation and its contents are about to be copied to the destination slot B[j] (which is in the INSERTING state) using a single compare and swap operation (line 24 in algorithm 3). The success of the compare and set operation depends on whether the contents of B[j] changed between the relocation. If the contents of B[i] get erased by a concurrent remove, then B[j] will still have the old content of B[i] due the successful compare and set operation. To prevent this scenario, after the key removal, the remove method also scans for slots in the INSERTING state. For each such slot, it simply puts a random number in the key section. This results in unsuccessful compare and set operations at the respective slot forcing any concurrent relocate to reload the updated contents of the slot it is relocating presently. For slots that are not taking part in relocation and whose key has been shuffled, the original key is restored eventually since a handle to the original slot entry holding the key is always used while updating its contents across slots (line 27 in algorithm 1).

Algorithm 4 Removing a key/value pair from the hash map.

```
1: procedure REMOVE(KEY)
2:     Slot ← H_f(x)
3:     for E in Slots[i], Slot <= i < Slot + H and [key]E = x and ([state]E = INIT or [state]E = REL) do ▷ Clearing the slot containing x.
4:         Slots[i].CAS(E,0)        ▷ Putting a random key inside slots in the INSERTING state to correctly serialize a relocate and the remove operation.
5:         if [state]E = INSERTING then
6:             Rand ← Random() & Rand.state = INSERT
7:             Slots[i].CAS(E,Rand)
8:         end if
9:         if isGrowing() then
10:            resize() and retry
11:        end if
12:    end for
13: end procedure
```

D. resize

We briefly outline the resizing strategy here. A resize operation doubles the maximal cardinality of the underlying array and rehashes all the keys to a new array. Recall that an extra state RES is used for a slot when it is about to be rehashed to a bigger array.

Firstly, the thread noticing the need for a resize iterates through all the entries in the hash table. For each slot, it remembers the old state it was in and updates its state to RES using a compare and set operation on the slot.

The interactions of the resize with the rest of the operations needs to be carefully calibrated to prevent the scenario where an operation being oblivious of the resizing may still work on

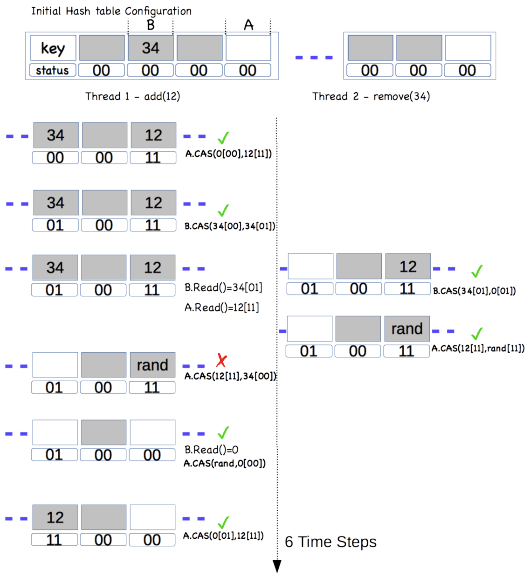

Figure 2: The figure depicts the scenario where the contents of a slot marked for relocation by one thread is removed by a second thread. The first thread inserts 12 at the empty slot `A` and putting it to `INSERTING` state. After this it marks `B` for relocation by putting it to the `REL` state and reads the contents of `A` to perform the relocation. Now the second thread comes and removes the key from `B` and shuffles the contents of `A` resulting in the failure of the compare and set operation performed by the first thread. This forces the first thread to re-execute the compare and set leading to the updated contents of `B` to be copied to `A`. Finally, contents of `A` is copied to `B` putting it in the `INSERTING` state leading towards a correct relocation.

the older stale copy of the hash table resulting in inconsistent behavior. These interactions per operation and how are they being handled by out algorithm are described below.

- `add()` **–:** The add method updates the hash table slots using multiple CAS operations. We couple the method `isGrowing()` along with each such CAS. This method returns true if the slot under consideration is in the `RES` state and false otherwise. For a thread executing the add method there can be two reasons for a CAS failure with respect to a slot. First, interference from an overlapping add or remove method and the second reason can be an interfering resize operation. In the former case the thread tries to re-execute the CAS by executing one more iteration inside the loop. For the latter case if the thread detects the slot to be in a `RES` state, it executes the resize function abandoning its present operation and retrying later when the resize is over.
- `contains()/remove()` **–:** These two method can know of a progressing resize by examining the present slot while they are iterating though their respective virtual bucket. If a slot in the `RES` state is discovered they proceed similarly as described for the add method above (Figure 4).

After each slot in the hash table is set to the `RES` state, a second iteration is executed. In this iteration, the thread rehashes all the keys and puts them to the `INIT` state inside a new array. The address of this new array can be retrieved

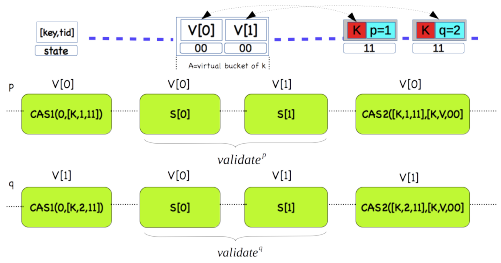

Figure 3: The sequence of operations performed by two threads during inserting of the same key k into a virtual bucket of size 2. The first compare and set (CAS_1) is executed when the new entry in the `INSERTING` state and containing k is placed into a slot in the virtual bucket during relocation. The second compare and set (CAS_2) is executed when validation returns true and k is made part of the hash map by restoring the value and clearing the state.

from a global look-up table accessible to all the threads. The slots that were in the `INSERTING` state are eliminated during the rehashing. These slots represents ongoing insertions, we make the add method retry its whole process after the resizing is over, so there is no need for these slots to be present in the new version of the hash table as these will be anyways introduced by the add during its second attempt. One interleaving worth outlining here is the interaction of an ongoing relocation with a resize. Recall that a relocation introduces two versions of the same key for a transient period of time. If a resizing takes place during this transient phase then the possibility of a duplicate during rehashing arises. This issue is prevented by letting the resize check for the presence of a key in the new hash table before blindly rehashing it. Finally the old hash map is discarded by routing its head pointer to the starting address of the new array using a single CAS instruction. Here head refers to the the starting address of the array housing the hash table.

V. Algorithm Analysis

This section contains proofs of correctness of our lock-free hopscotch algorithm and the progress guarantees of the operations on the hash map. Our model of multiprocessor computation follows [4] and we use operational style arguments. We prove that our algorithm compiles with the abstract set semantics. We make use of the sequential specification of a "dynamic dictionary" as defined in [1]. Given this sequential specification, we mark specific linearization points mapping operations in our lock-free codes to a sequential execution.

A. Progress Conditions

A method is lock-free if there there exists a thread that makes progress once in a while ensuring that at-least one thread is guaranteeing system wide progress.

Lemma 1: The relocate operation finishes after a finite number of steps, or another operation must have executed one step of its execution guaranteeing system wide progress.

Proof: After a slot `i` is paired with a slot `j` for relocation using the `REL` state, the relocate operation tries to physically swap the contents of both the slots. This swapping might fail

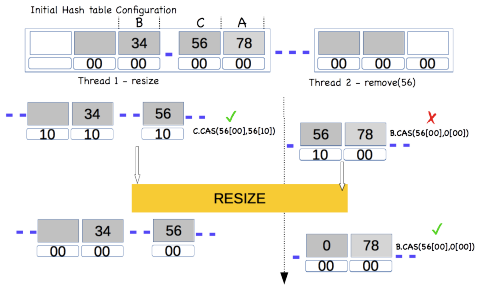

Figure 4: The figure shows an overlapping resize and remove method. After the slot C is marked for resize the CAS in the remove method fails for the second thread. The second thread now being aware of a ongoing resize joins the effort in resizing the hash table. After the resizing is done and all the slots brought to their original state, the second thread retries the removal. This time it correctly removes the key from slot.

due to interference caused by a remove or a resize operation. The former case implies that a remove has made progress in its execution and has moved a step in direction of its completion guaranteeing system progress. In the case of a resize, the swapping fails because the resize has marked either slots for rehashing. In this case also, the thread executing the relocation goes and executes the resize operation instead, thus guaranteeing progress. ■

Theorem 2: The insertion operation finishes in a finite number of steps, or else at-least one another operation must have executed one step of its execution guaranteeing system wide progress.

Proof: The add method first linear probes for a new empty slot to place the key. Other operations can interfere with this probing by changing the contents of the slot that was read to complete the initial placement of the new key. This can happen in one of two ways. An overlapping `add()` has tampered with the contents of this slot or a resize has marked the slot for rehashing. In both these case either the hash table has progressed one step in the direction of a new insertion or a resize. After this the add method executes the relocate method which also guarantees progress (see lemma above). Now based on the return value of validate method, the `add()` executes the CAS at line 27 or at line 34. Either of these CAS may fail because of a overlapping resize, `add()` or a `remove()`. The overlap with an `add()` occurs when it is being executed by a thread with a lower thread identifier on the same key. The `remove()` overlaps when it places a random value in the slot (line 7 in remove()). In any case, either the overlapping `add()` or the `remove()` is making progress. ■

Theorem 3: In the absence of a resize the `contains()` and `remove()` methods always finishes within `H` number of steps.

Proof: This can be easily proved by reasoning about the loops bounded by the hopscotch range `H` present inside both the methods. ■

B. Correctness

Lemma 4: The validate method correctly maintains atmost a single copy of the key x in the hash map in the presence of multiple insertions of x by different threads.

Proof: Without loss of generality let the size of a virtual bucket in our hash map be 2. Two threads, $\mathtt{t}_1$ and $\mathtt{t}_2$, with identifier values `p` and `q` (`p` < `q`), are trying to place the key `k` into its virtual `v` at slots `v[0]` and `v[1]` respectively, see Figure 3. The notation $\mathtt{o}^{\mathtt{t}}[\mathtt{true/false}]$ denotes that the operation `o` executed by a thread `t` returns true/false. If `o` could not be executed by `t` due to interference by other threads then the return value of `o` is set to false. We use, $\mathtt{o}_1 \rightarrow \mathtt{o}_2$, to represent the fact that the operation $\mathtt{o}_1$ takes effect before $\mathtt{o}_2$ on the hash map. We define an operation $\mathtt{S[i]}^{\mathtt{t}}$ with respect to `k` such that:

$$S[i]^t[true] \iff v[i].key \neq k \vee Mark[i]^t[true]$$
$$S[i]^t[false] \iff v[i].key = k \wedge v[i].tid < t \vee Mark[i]^t[false]$$

The $\mathtt{Mark[i]}^{\mathtt{t}}[\mathtt{true/false}]$ operations attempts to place a marker at `v[i]` and returns true on successful placement and false otherwise. Recall from the description of the validation step, that for placing the marker the slot should contain the key `k` with a thread identifier greater than `t`. Now, for the duplication of `k` the following holds (refer to figure 3 for the description of the CAS operations):

$$v[0].key = k \wedge v[1].key = k \implies CAS_1^p[true] \rightarrow CAS_2^p[true] \wedge CAS_1^q[true] \rightarrow CAS_2^q[true]$$

We prove by contradiction that due the presence of the validate step during insertion, the right size of the implication can not be true, therefore only one copy of `k` can remain in either `v[0]` or `v[1]`. From definition of validate:

$$validate^t[true] \implies \forall i = [0,n] S[i]^t[true] \quad (1)$$
$$validate^t[false] \implies \exists i = [0,n] S[i]^t[false] \quad (2)$$

w.r.t both the threads either (3) holds or (4) holds.

$$CAS_1^p[true] \rightarrow CAS_1^q[true] \quad (3)$$

$$CAS_1^q[true] \rightarrow CAS_1^p[true] \quad (4)$$

Assuming (3) holds the following is satisfied:

$$validate_1^q[false] \implies CAS_2^q[false] \quad using(2) \quad (5)$$

Assuming (4) holds two cases arise:

$$1: S[2]^p[true] \implies validate^p[true] \implies CAS_2^q[false] \quad (6)$$
$$2: S[2]^p[false] \implies validate^p[false] \implies CAS_2^p[false] \quad (7)$$

Equations (5), (6) and (7) are all contradictions. Therefore, only a single copy of k can be present in the hash map. The other accompanying proof of emptiness i.e.:

$$v[0].key \neq k \wedge v[1].key \neq k \implies CAS_1^p[true] \rightarrow CAS_2^p[false] \wedge CAS_1^q[true] \rightarrow CAS_2^q[false]$$

can be proved using a somewhat similar contradiction technique and is not presented here due to space limitation. ■

Theorem 5: The add() and remove() methods are linearizable.

Proof: A key x can be considered part of the hash map only when it lies inside its virtual bucket and the slot containing it is in the `INIT` state. The relocate method inside the `add()` is responsible for transporting a key whose probe sequence is greater than `H` into its virtual bucket. Successful execution of the compare and set at line 27 in the add method completes the insertion of x. This is the linearization point for the add method. A successful/unsuccessful `contains(x)` operation overlapping with an `add(x)` can be linearized after/before the `add(x)`, if it read the contents of the slots to which x gets mapped before/after the successful execution of the CAS operation (linearization point of the add() method). For multiple overlapping instances of the `add()` operation, inserting the same key x into the hash map, only one `add()` is successful in inserting the key due to the presence of the validation step (see lemma 4 above). Therefore all the unsuccessful `add()` operations, can be linearized after the successful `add()`.

A successful `remove()` is linearized after it erases the actual key from the associated table entry. The CAS at line 4 in the `remove()` method physically removes the key, hence this is the linearlization point of the `remove()`. Due to displacement of a key in the `add()`, a key can be present at two entries in the hash table. In this case a concurrent `remove()` and `contains()` method can be linearized based on which method atomically erased/read the last entry containing the key. The `add()` and `remove()` is linearized based on which method successfully executed their respective compare and set operation, assuming the key is present within its virtual bucket. An interesting case is when an `add()` overlaps with a concurrent `remove` involving the same key which is already present in the hash table. The add method is either successful or unsuccessful in inserting the key based on whether the validate step inside the `add()` read the contents of the slot containing the key after or before linearization point of `remove()`, in which case it can be linearized after or before the `remove()` method respectively. ■

Theorem 6: The contains() method is linearlizable.

Proof: A successful `contains()` is linearized when it finds the searched key in the hash table. An unsuccessful `contains()` can occur after it failed to find the key inside its virtual bucket. The `contains()` can be linearized to one of these operations, unless there was an addition of the searched key by some concurrent `add()` while the scan in the `contains()` was in progress. In which case the `contains()` can linearized after/before the `add()` based on where the it scanned the slot being modified by the `add()` after/before execution of the linearization point of the `add()`. The same can be argued about the overlap of a `contains()` with the `remove()` method. Also, the `contains()` is immune to displacements as it always scans all the `H` locations in the hopscotch range. ■

VI. Experimental Framework and Results

This section empirically compares the performance of our lock-free hopscotch hash table to the most efficient prior concurrent hash tables. For experimentation, we use the standard micro-benchmark technique similar to the work in [3].

A. Experimental Framework

In our benchmarks each data point was sampled 10 times and the average was plotted. The benchmark results were collected for a two socket `Intel i7-5960`, 16 core server CPU running at 3.00 GHZ with 20 MB of last level cache (LLC) and 32 GB of RAM. Our hash map (`LF-Hop`), with hopscotch range $H = 32$, was compared with the following effective concurrent hash tables that exists in literature.

- `Locked-Hop`: This is the original lock based hopscotch hash table proposed by Herlihy and Shavit [3] with $H = 32$.
- `Cuckoo`: We used a recently proposed lock-free version of the Cuckoo hash by Nguyen [8].
- `Split-Ord`: We used an optimized version of the more popular dynamically resizeable closed addressed hash table by Shalev and Shavit [10].

To make sure the hash tables did not completely fit in the LLC, we chose a table size of 2^{23} randomly generated unique 64 bit key-value pairs. For the experiments, the size of the hash tables, except the `Split-Ord` hash, was pre-allocated towards maintain a load factor of 0.4. The degree of concurrency was varied from 2 to 64 threads (4 times the maximal number of actual hardware threads supported by the machine). All tested hash maps were implemented in C++11 compiled using `GCC 5.4` with the -O2 flag set on an `Ubuntu-16.0` system.

B. Benchmark Results

We begin by comparing the hash tables for throughput measured in terms of million queries (insert, delete, search) processed per second (MQPS). Firstly, the hash tables were tested on two types of scenarios. The first one is an insertion only scenario where the threads dumped a total of 2^{23} items (every thread dumped an equal amount) in the respective hash table. As can be seen in figure 5, our hash table achieves

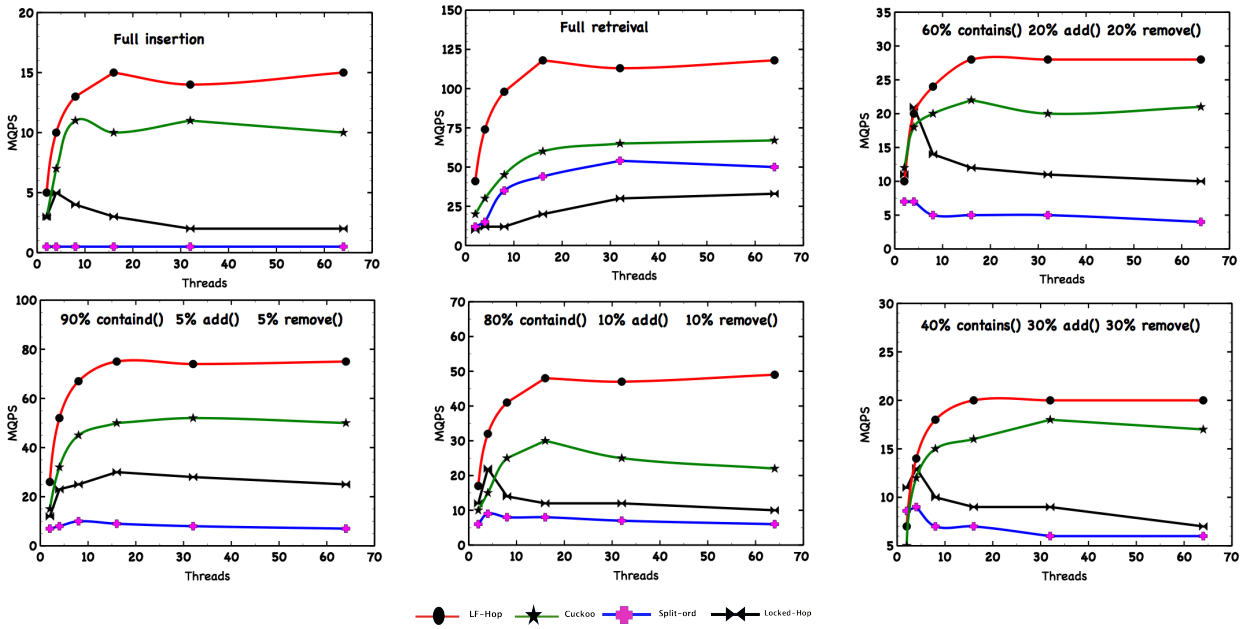

Figure 5: Comparison of the throughput of our hash table with other lock-free hash tables under various operation mixes.

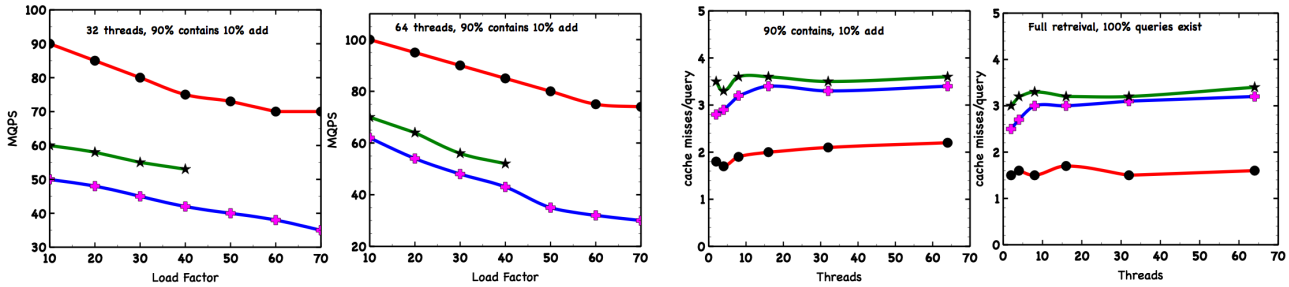

Figure 6: The above graphs present the variation of query throughput with changing load factor values and the number of cache misses per operation incurred by each hash table respectively.

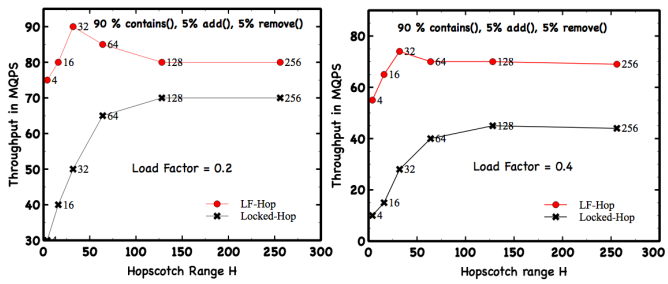

Figure 7: Comparison of the throughput of our hash table with varying values of H on load factor values of 0.2 and 0.4 with 64 threads respectively.

1.5 to 2 times higher throughput compared to the `Cuckoo` and `Locked-Hop` hash table respectively. The dynamic re-sizing overhead in the Split-Ord hash table lowers its throughput, which was absent in the other hash tables because of pre-allocated storage. Next, we execute 8 M successful search queries on each hash table. As can be seen, our hash table processes twice the number of queries compared to the `Cuckoo` hash table. Also, our hash table scales well with the increasing number of threads compared to the other hash tables which flatten out early. This high key retrieval rate can be attributed to the sequential and limited memory accesses in our design. Now we experiment with various query mixes. We analyzed the general throughput of the commonly found distribution of actions: 90% contains(), 5% add(), and 5% remove() and the less common ones of [80-60-30]% contains(), [10-20-30]% add(), and [10-20-30]% remove(). Our hash table achieves 1.5-2 times higher MQPS compared to the `Cuckoo` and the `Split-Ord` hash tables respectively. With the increasing percentage of insertions and deletions, our hash table maintain its scalability unlike the `Locked-Hop` hash table achieving comparable throughput

with lesser threads but deteriorating as the number of threads increase.

Next we study the sensitivity of the hash tables towards changing table densities (load factors) (figure 6). We declare failure upon noticing the need for a resize of the respective hash table. Due to the linear probing and the chaining strategy, our hash table and the `Split-Ord` table succeeds at high load factors in contrast to the `Cuckoo` hash table failing at load factors beyond 0.5. With higher load factors the throughput of our hash table decreases. This due to the fact that the probe sequences becomes longer as the table becomes denser. Also, at higher load factors, the numbers of relocations inside our hash tables increases.

We now analyze the cache behaviour of our lock-free hopscotch hash table. A contains() and remove() operation incurs $\frac{H}{B}$ number of cache misses, `B` is the cache block size, in the worst case. The number of cache block misses incurred during an add() operation depends on the length of the probe sequence. Additional cache misses are incurred in case of a relocation. Due to the sequential scan of memory addresses both in insertion and relocation of a key, the cache misses are reduced. A measurement of number of cache misses of the lock-free hash tables is presented in figure 6. Our hash table triggers approximately 1.5 cache misses per operation which is almost 2 times lower compared to the `Cuckoo` and the `Split-Ord` hash tables respectively.

Finally, we study the variations in throughput of our hash table with respect to the hopscotch parameter, `H`, see figure 7. This is an important parameter as it fixes the size of a virtual bucket. We compare our implementation with the `Locked-Hop` hash table. As we resort to sequential scanning of a virtual bucket in our algorithm compared to the hopping method used in the `Locked-Hop` hash table, a slight decrement in throughout is expected. The throughput of our hash table increases initially and peaks at H=32, at which point the virtual bucket fits in a cache line. For higher H the throughput decreases and saturates later. On the other hand, the `Locked-Hop` peaks at a slightly higher value of H and saturates afterwards.

VII. Conclusion

In this work, we described a lock-free hopscotch hash table that improves upon the existing lock-free hash tables. Unlike the Split-Ord hash table that stored the keys in a lock-free link list, our hash table stores the keys in an array thus improving upon the total number of cache misses incurred while processing queries. The more recently introduced lock-free `Cuckoo` hash table has a relatively more complicated look-up procedure with counters attached to array indices and a two round query protocol that is more prone to failures. Also, the relocation path has no well-defined memory access pattern and can be very long. On the other hand, our hash table has a very straightforward and simple look-up procedure with zero failure probability. The relocation has sequential memory access pattern which proves better with respect to caches and is bounded by the length of the probe sequence. The leader election algorithm and the signaling mechanism using specific markers can be used as general techniques to coordinate threads operating on the same region inside a memory block. We plan to integrate our hash table inside real world applications like databases to further testify its effectiveness. We also plan to port our design to more parallel hardware, for example GPU's, to test its scalability.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[2] Michael Greenwald. Non-blocking synchronization and system design. Technical report, Stanford, CA, USA, 1999.

[3] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *22nd Intl. Symp. on Distributed Computing*, 2008.

[4] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[5] D. Lea. Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package.

[6] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004.

[7] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[8] Nhan Nguyen and Philippas Tsigas. Lock-free cuckoo hashing. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 627–636, 2014.

[9] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[10] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.