An Efficient Wait-free Resizable Hash Table

Panagiota Fatourou
FORTH ICS & University of Crete,
Department of Computer Science
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
FORTH ICS
nkallima@ics.forth.gr

Thomas Ropars
Univ. Grenoble Alpes
thomas.ropars@univ-grenoble-alpes.fr

ABSTRACT

This paper presents an efficient wait-free resizable hash table. To achieve high throughput at large core counts, our algorithm is specifically designed to retain the natural parallelism of concurrent hashing, while providing wait-free resizing. An extensive evaluation of our hash table shows that in the common case where resizing actions are rare, our implementation outperforms all existing lock-free hash table implementations while providing a stronger progress guarantee.

ACM Reference Format:
Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas Ropars. 2018. An Efficient Wait-free Resizable Hash Table. In *SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3210377.3210408

1 INTRODUCTION

As the core count is increasing in modern processors, designing data structures that provide good performance when a large number of threads access them concurrently is a must [22], but it is also a highly challenging task [10], especially if strong liveness guarantees, such as *wait-freedom*, are to be provided [13]. Wait-freedom is highly desirable as it ensures *progress* for all running threads independently of their speeds or any *crash* failures that other threads may experience. Specifically, wait-freedom ensures that *every* operation on the data structure (executed by a running thread) will complete within a finite number of steps. *Lock-freedom* is a weaker progress condition which guarantees that *at least one* thread makes progress, thus allowing other threads to starve.

A hash table is a data structure commonly used to implement a dictionary of key-value pairs. It provides two *update* operations (Insert and Delete) and a LookUp operation. A hash function is used to associate keys to buckets so that each operation on the hash table takes constant average time [14]. To ensure this property even when the number of stored items varies over time, dynamic hashing aims at dynamically resizing the hash table to adapt the number of buckets to the number of items [4]. Resizing actions (splitting or merging of buckets) are triggered during Insert and Delete operations.

It is commonly acknowledged that, in most cases, LookUps are by far the most frequent operations on a hash table [14, 23]. Given that update operations are not frequent, resizing actions are rare events since the number of items to store does not vary much over time. In theory, hashing can be made very efficient in a concurrent environment due to its *natural parallelism* [14]. In most cases, concurrent operations access different parts of the hash table, and thus they can proceed in parallel without any interference with one another. However, the case of dynamic hashing is more complex. Allowing hash table operations to be executed in parallel with resizing actions while ensuring *linearizability* [15], is difficult, especially for algorithms providing strong progress guarantees [19, 21].

This paper presents a new wait-free implementation of a resizable hash table. Our hash table aims at achieving best performance for the most common operations on a hash table, while providing *wait-freedom* progress guarantee. To guide our design, we identify two design rules that are important to achieve high performance at large core counts: (A) LookUp operations should always be allowed to proceed without any synchronization; (B) when no resizing actions are executed, update operations applying to different buckets should be allowed to progress fully in parallel (i.e., without any interference with each other). These rules aim at preserving the natural parallelism of hash tables in the most frequent case where no resizing is required. Rule (A) additionally aims at minimizing the cost of the most frequent operations in all cases.

To implement a hash table that complies with these design rules, we propose an algorithm based on *extendible hashing*, a dynamic hashing technique that considers keys as bit strings [5]. An extendible hash table can be seen as an array (the *directory*) of pointer to fixed-size buckets. In its sequential implementation, every resizing operation is local, e.g., one bucket can be split into two without modifying the other buckets.

Our wait-free extendible hash table relies on the PSim universal construction [7]. PSim provides a general mechanism to implement any concurrent object in a wait-free manner. It exploits the well-known technique [6] of having a thread that executes an operation help other announced operations by applying them, in addition to its own, on a local copy of the simulated object state. Then, it attempts to change a shared reference to the object state to point to this local copy. PSim results in highly-efficient wait-free implementations of data structures that have a single or a small number of points of contention, such as stacks and queues [7]. In this paper, we show how we can use several instances of the PSim algorithm and synchronize them appropriately to get an efficient wait-free implementation of an extendible hash table. To match our design rules, our hash table uses an instance of PSim's algorithm for each bucket. These instances run update operations on each bucket fully independently as long as no resizing actions are required. An additional instance of PSim's algorithm is used to manage resizing actions modifying the state of the hash table. The crux of our algorithm is in the mechanisms used to coordinate the different instances of PSim during resize actions in order to ensure linearizability and wait-freedom, while complying with our design rules.

SPAA '18, July 16–18, 2018, Vienna, Austria

We evaluate our algorithm experimentally and compare it with that of two state-of-the-art lock-free concurrent hash tables: the lock-free resizable hash table proposed by Liu et al. [19] and the hash table based on a *split-ordered list* proposed by Shalev and Shavit [21]. Experiments run on two Intel processors (48-core Haswell, and 64-core Broadwell) show that in a directory-stable state, i.e., when resizing actions are rare, the performance of our algorithm is highly competitive. When manipulating a small number of items, and when the percentage of LookUps is high, our wait-free algorithm outperforms the most efficient lock-free algorithm by up to 47% on a 64-core machine. When the size of the hash table increases, a modified version of the *lock-free* algorithm presented by Liu et al. [19], that we propose, becomes the most efficient with our algorithm being the second most efficient. The very high performance of our algorithm in directory-stable states comes at the cost of slower resizing actions. However, our experiments demonstrate that the resizing cost is acceptable when it can be amortized over long runs. In this case, our wait-free hash table is as efficient as or more efficient than the best existing lock-free solutions.

The main contributions of this paper are summarized as follows:

- We present a new wait-free resizable hash table based on extendible hashing (Section 4). Its implementation follows two design rules that aim at preserving the natural parallelism of dynamic hashing for the most frequent operations.
- We provide an extensive performance evaluation of the new algorithm at large core counts that shows that, when resizing actions are rare, our wait-free hash table largely outperforms all previous lock-free implementations (in terms of throughput), at the cost of slower resizing (Section 6).

The results of our experiments show evidence that the design rules we identify are of key importance to build efficient non-blocking resizable hash tables.

2 RELATED WORK

Hash tables are important data structures in domains ranging from operating systems' kernels [1, 23] to runtime and programming languages [16].

The easiest way to implement a resizable hash table is probably using locks, i.e., in a *blocking* way. One such implementation is the *ConcurrentHashMap* provided in the Java `concurrency.utils` library [18]. It uses a fixed number of locks, each of which guards a subset of the buckets. Resizing can be performed only by a thread that has acquired all locks, thus excluding all other threads from executing operations during a resizing phase. A concurrent *extendible* hashing algorithm based on locks was presented by Ellis [3]. It is based on a two-level locking scheme where a lock on the directory must be grabbed first, before locking a specific bucket.

Non-blocking resizable hash tables are appealing because of their stronger progress guarantees that have been shown to lead to higher performance in several studies [10, 21]. One of the first lock-free non-resizable hash table implementations was described by Michael [20]. It is based on an array of lock-free linked lists. At the same period, Greenwald [11] presented a lock-free resizable hash table that relied on a DCAS (*double-compare-and-swap*) operation. However, DCAS is not supported on most hardware architectures.

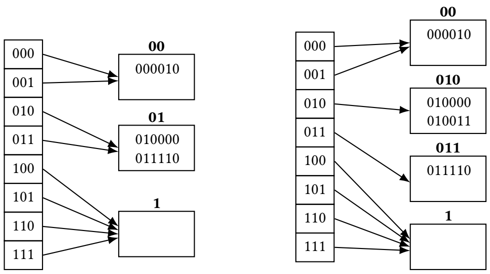
The lock-free resizable hash table proposed in [21], which we will call LF-Split, relies on a *split-ordered* list. The items of the hash table are stored in an ordered linked-list and a separate array of pointers pointing to elements in the linked list plays the role of the directory. The items belonging to bucket numbered *i* are all the items accessible in the linked-list starting from the node pointed to by entry *i* of the directory and finishing at the node pointed to by the next entry of the directory. Hence, inserting a new element in a bucket simply requires to insert it in the linked list. Splitting a bucket into two only implies adding in the directory a new pointer to an element in the list. Relying on a list to implement the hash table makes resizing operations very efficient. However, LookUp operations might be less efficient than with array-based hash tables because of the cost of pointer chasing paid when iterating over the elements of a bucket [2, 19]. LF-Split complies with neither of the design rules we introduce to optimize directory-stable-state performance. Indeed, rule (A) is violated because when a node is marked for deletion, threads running LookUp operations might have to help removing these nodes from the list. Moreover, a global counter has to be updated when an item is removed or a *new* item is inserted, which breaks rule (B). The experiments presented in Section 6 show that these two facts limit the performance of LF-Split when the directory is stable.

To the best of our knowledge, only two wait-free resizable hash tables have been described thus far [8, 19]. The algorithm of Feldman et al. [8] is based on a multi-level array resulting in pointer chasing and therefore reduced performance for LookUp operations.

Liu et al. have proposed in [19] an array-based resizable lock-free hash table, which we will call LF-Freeze. Buckets are implemented as arrays of items, and the directory is an array of pointers to items. When the directory needs to be resized, the buckets are *frozen*: no update operations are allowed to proceed on these buckets anymore. Splitting the buckets is then lazily done during Insert operations. LF-Freeze respects the two design rules that we have defined. The experimental analysis on large multicore machines that is provided in [19], shows that LF-Freeze outperforms LF-Split [21]. Our algorithm shares some of the design ideas of LF-Freeze. However, it ensures wait-freedom by using instances of the PSim algorithm for ensuring synchronization.

Liu et al. also proposed in [19] a wait-free variant of their lock-free implementation, which we will call WF-Freeze. In this variant, threads should help each other running update operations and resize actions on the buckets. This is done by assigning sequence numbers to all update operations, a technique that is implemented using a globally shared counter (thus violating rule (B)). The experiments in [19] show that WF-Freeze exhibits performance that is by far lower than the performance of LF-Freeze. Our algorithm ensures wait-freedom without using any globally shared counter.

Kogan and Petrank proposed in [17] the *fast-path-slow-path* technique, an approach to get a wait-free variant of a lock-free algorithm with a relatively small performance cost. In their experimental analysis, Liu et al. included a wait-free variant of their lock-free algorithm based on this technique. The resulted algorithm performs much better than WF-Freeze (their brute-force wait-free solution), but its performance is lower than that of LF-Freeze (their lock-free algorithm). Our evaluation shows that our brute-force



(a) Initial state **(b) After inserting** 010011

Figure 1: An extendible hash table. The directory has a depth of 3 and each bucket can store at most 2 items.

wait-free implementation outperforms LF-Freeze in several cases when the hash table is in a directory-stable state.

3 EXTENDIBLE HASHING

An *extendible hash table* is a data structure of two levels. It consists of a set of *buckets*, in each of which a fixed number b of items are stored, and a resizable array, called the *directory*, where each entry stores a pointer to a bucket. Extendible hashing manipulates hash keys as bit strings: the bit strings are used to distribute items to buckets. The directory has 2^D entries, where D is a parameter called the *depth* of the directory. Figure 1 describes an extendible hash table with $D = 3$. A prefix corresponding to the D most significant bits of the hash key is used to associate items to directory entries and the buckets they point to. For instance, in Figure 1, key 010000 is associated with the directory entry 010. The depth of the directory is at most as big as the total number of bits in the hash key. The size of the directory is always exponential to its depth.

In practice, the number of buckets does not have to be the same as the number of directory entries: to improve memory efficiency, it is adapted to the number of items to store. In Figure 1a, a single bucket is allocated to store all keys with prefix 1 since there is no such key, and all four directory entries with prefix 1 point to that bucket. The *depth* of this bucket is 1, i.e., it equals the length of the prefix that identifies the hash keys that are to be stored in it.

As illustrated in Figure 1b, a bucket is split in two buckets when it is full (i.e., when it already contains b items) and a newly inserted item must be stored in this bucket. For instance, to insert key 010110, two new buckets 010 and 011 should be created to replace the existing bucket 01. In this case, the directory will not be resized because the bucket with prefix 01 has depth 2 and the two newly created buckets to replace it, have depth 3 which is smaller than or equal to the directory depth. However, to insert another item 010111 in the table of Figure 1b, resizing the directory would be required. The depth of the directory should be increased to 4 to allow storing the pointers of two new buckets 0100 and 0101. We remark that the resizing actions are *local*: to replace a bucket with two new buckets, all the items of the old bucket are stored in the new buckets and no additional elements are stored in them.

4 A WAIT-FREE IMPLEMENTATION OF AN EXTENDIBLE HASH TABLE

This section presents the new wait-free implementation of the extendible hash table. We start by describing the main ideas of the algorithm. Then, we present how the hash table works when no resizing occurs. Finally, we provide the details of resizing. Our wait-free algorithm incorporates and builds upon the code of PSim [7] for updating the buckets and the directory.

In our implementation (as well as in the experimental analysis of Section 6), an invocation of Insert for an already existing key, updates the value associated with the key. This semantics correspond to that of dictionaries provided by popular programming languages such as Java[1] or Python[2]. The description assumes that a wait-free garbage collector is available. (Section 5 provides a discussion on memory reclamation.) We consider a system of n threads.

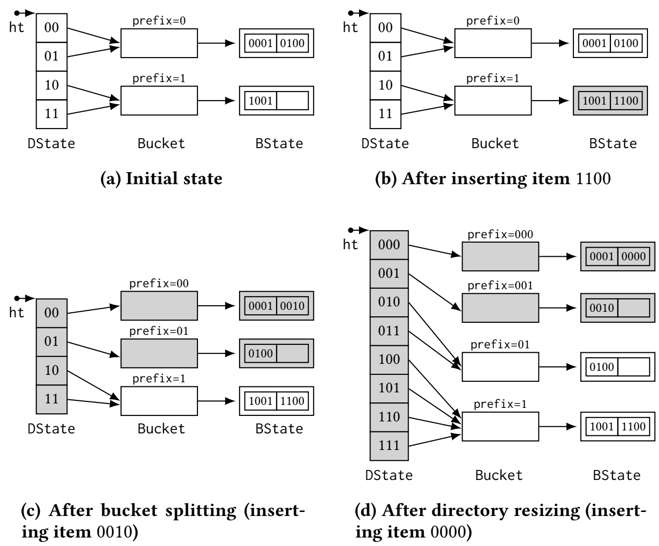
4.1 The algorithm in a nutshell

Figure 2 presents the structure of our hash table and describes how it evolves when update operations are executed. The main challenge when designing a resizable hash table is allowing operations of the hash table to be executed in parallel with resizing actions. To this end, our algorithm uses two levels of indirection between the DState object that implements the directory and the BState objects that store the items of buckets (see Figure 2a). In a BState object, items are stored in a fixed size array (in Figure 2, this size is 2). The data records used to implement our hash table are presented in Figure 3. To distinguish a variable that stores a reference to an object, we add the suffix _p to its type.

To insert a new element into a non-full bucket in a wait-free manner, a thread creates a local copy of the corresponding BState. It applies its operation on this local copy and uses a CompareAndSwap (CAS) operation to attempt to update the BState pointer in the corresponding bucket to point to its local BState object (see Figure 2b). If the corresponding bucket is full, it will be replaced by two new buckets to complete the Insert. This case is illustrated in Figure 2c. The directory, i.e., the DState object, must be updated to store references to the new buckets. To perform the update on the directory state in a wait-free manner, a thread first creates a local copy of the currently active DState object (pointed to by the shared variable ht). Note that this copy contains pointers to the existing buckets. The thread can then create the new buckets from the full bucket (buckets 00 and 01 from bucket 0 in Figure 2c), update its local DState object accordingly, and finally, try to make its local DState object the active DState object by updating ht using CAS.

The algorithm has to ensure that update operations are not lost if they run concurrently with a resizing action that requires replacing the current DState object. Thus, our algorithm has to handle the following two cases that might arise during the creation of a new DState object: (i) no update should be lost if a pointer to a bucket is replaced with pointers to new buckets in the new DState object, and (ii) no update targeting the non-full buckets should be lost, i.e., the non-full buckets must still be referenced by the new DState object. To prevent the appearance of the first case, our algorithm ensures that (1) a bucket is split only if it is full and (2) no update

[1] Package java.util.AbstractMap
[2] https://docs.python.org/3/library/stdtypes.html#dict



(a) Initial state

(b) After inserting item 1100

(c) After bucket splitting (inserting item 0010)

(d) After directory resizing (inserting item 0000)

Figure 2: Operations on the wait-free extendible hash table. (The new objects created during an operation are grayed)

```
struct Operation:
    type : {INS, DEL}
    key : integer
    value : integer
    seqnum : integer

struct Result :
    status : {TRUE, FALSE, FAIL}
    seqnum : integer

struct DState :
    depth : integer
    dir[2^depth] : Bucket_p

struct Bucket:
    prefix : bit−string
    depth : integer
    state : BState_p
    toggle[n] : boolean

struct BState :
    items : set of key−value pairs
    applied[n] : boolean
    results[n] : Result
```

Figure 3: Data structures definitions (for *n* threads).

```
Shared variables :
    ht : DState_p
    help[n] : Operation_p

Persistent private variable :
    opSeqnum_i : integer
```

Figure 4: Variables for thread T_i.

operation (not even Delete) is executed on a full bucket. To prevent the second case from occurring, our algorithm employs two levels of indirection between a `DState` object and the `BState` objects. Assume that a `BState` object is updated as shown in Figure 2b, while the `DState` object pointing to this bucket is also updated (as shown Figure 2c). In this case, the new `DState` object will still point to the previously existing `Bucket` objects that are not split (i.e., that are not full). Thus, even if the `BState` reference stored in a `Bucket` changes, it will still be accessible through the new `DState` object.

Resizing the directory is required when its depth becomes smaller than the depth of new buckets that are created during an Insert operating on a full bucket. In this case, after creating a local copy of the currently active `DState` object (i.e., after copying all pointers to buckets locally), the thread executing the Insert must create a new `DState` object of larger depth, and copy the existing bucket references into it. Figure 2d illustrates this case.

In the following sections, we provide the details of our algorithm. We discuss how update operations on `DState` and `BState` objects are implemented in a wait-free manner, as well as how they are synchronized to ensure linearizability.

4.2 Data structures for the resizable hash table implementation

Recall that the data records used to implement our hash table are presented in Figure 3. Part of the `Bucket` state is stored in a `BState` record. This part is copied by every thread that wants to apply an atomic update on the bucket. Each `Bucket` record, as well as each `BState` record includes a bit vector of size *n* (called `toggle` and `applied`, respectively) that are used to efficiently track the pending operations on a bucket. A `BState` record also stores an array, called `Result`, that is used to store the results of active operations. (We describe how these fields are used in more detail in the subsequent sections.) The fixed size array[3] used to store the items associated with a `BState` is called *items*. In the following, we use the notation *items[key]* to set or get the value associated with one key. The hash table is represented by a record of type `DState` that is composed of an array of references to elements of type `Bucket` and a variable `depth` storing the depth of the directory.

The shared variables and the private persistent variables for each thread are provided in Figure 4. Each operation initiated by a process *i* is announced in element *i* of the shared array *help*. A per-thread sequence number ($opSeqnum_i$) is introduced but it will only become useful when discussing resizing in Section 4.4.

4.3 The case of no resizing

This section describes how Insert and LookUp work when no resizing is required. Figure 5 provides code for Insert and LookUp. Delete is implemented in the same way as Insert.

As described in Section 4.1, to insert an item in a non-full bucket *b*, a thread must update the corresponding `BState`. This is done in `ApplyWFOp()`. Note that the thread executes other pending updates on the bucket (in addition to its own), leveraging the core idea of `PSim`. Specifically, the thread first announces its operations (lines 40 and 50). Then it tries to apply its operation, as well as all other pending operations for bucket *b*, on a local copy *newb* of the bucket state (lines 52-60). Finally, it executes a `CAS` to make this updated state the new value of *b.state* (line 61). When a thread executes an operation on behalf of another thread, it stores the result of this operation in *newb.results* so that the other thread can find there the result of its operation (line 47).

[3] The size of the array equals the maximum number of elements a bucket can store. To look for a key, a thread has to iterate over all elements currently stored in the array.

```
33 def Lookup(integer key): (boolean, integer)
34     htl ← ht ★
35     bs ← htl.dir[Prefix(key, htl.depth)].bstate
36     return ((key ∈ bs.items) ? (TRUE, bs.items[key]) : (FALSE, −1))

38 def Insert(integer key, integer value): boolean
39     opSeqnum_i ← opSeqnum_i+1 ★
40     help[i] ← new Operation(INS, key, value, opSeqnum_i)
41     htl ← ht ★
42     ApplyWFOp( htl.dir[ Prefix(key, htl.depth) ] )
43     htl ← ht ★
44     if htl.dir[ Prefix(key, htl.depth) ].state.results[i].seqnum != opSeqnum_i: ★
45         ResizeWF() ★
46     htl ← ht ★
47     return htl.dir[ Prefix(key, htl.depth) ].state.results[i].status

49 def ApplyWFOp(Bucket_p b):
50     Flip(b.toggle, i) # change the value of bit i
51     for k in 1..2:
52         oldb ← b.state
53         newb ← New BState(oldb) # create a copy of object oldb
54         t ← b.toggle
55         for j in [ tid for tid in 1..n if t[tid] != newb.applied[tid] ]:
56             if newb.results[j].seqnum < help[j].seqnum: ★
57                 newb.results[j].status ← ExecOnBucket(newb, help[j])
58                 if newb.results[j].status is not FAIL: ★
59                     newb.results[j].seqnum ← help[j].seqnum ★
60         newb.applied ← t
61         CAS( b.state, oldb, newb)

63 def ExecOnBucket(BState_p b, Operation_p op): status
64     if b.items is full:
65         return FAIL
66     else:
67         exist ← (op.key ∈ b.items)
68         if op.type == INS:
69             b.items[op.key] ← op.value # updates or inserts
70             return !exist
71         else:
72             delete b.items[op.key] # does nothing if key not present
73             return exist
```

Figure 5: Code of Insert and LookUp operations (lines marked with the ★ symbol are required only to implement resizing).

To identify pending updates on a bucket, the `toggle` vector of the `Bucket` and the `applied` vector of the corresponding `BState` are used. To announce an update operation *op* on a bucket b, a thread T_i flips its bit in $b.toggle$ (line 50) in an atomic way. Hence, if a bit j has different values in $b.toggle$ and $b.state.applied$, then thread T_j has a pending operation. Note that T_i announces *op* in $help[i]$ before flipping its bit in $b.toggle$.

After T_i has executed the body of the loop on lines 51-61 twice, it is guaranteed that the operation of thread T_i has been executed [7]: If both T_i's `CAS` operations fail (line 61), it means that another thread T_j has run an iteration of this loop and has executed a successful `CAS` between T_i's first and second `CAS` step. Since an iteration executes all pending operations and thread T_i had registered its own operation before executing its first `CAS` step (line 50), T_j has applied the operation of thread T_i. So, after executing it second `CAS` step, T_i can read the result of its operation in the corresponding entry of the *result* array of the active `BState` (line 47). The function `ExecOnBucket()` (lines 63-73) is called to (sequentially) execute an update operation on a local copy of a bucket state.

Complying with design rule (A), LookUp operations (line 33) do not require any synchronization, i.e., they execute a code that is exactly the same as their sequential code independently of whether the buckets they are directed at are being resized when they are executed. Doing so does not violate linearizability because when a thread makes a copy of the current `BState` to apply an update to it, it copies the set of items as part of this state. Therefore, the bucket states are immutable, allowing to access them safely without synchronization. Performing this copy has a cost for update operations. However, it allows for very efficient LookUp operations which are the most frequent operations performed on a hash table.

To access a bucket (e.g., line 35), a thread first stores a copy of *ht* in local variable *htl* (lines 34, 41, 43, and 46). This is needed because $ht.dir$ and $ht.depth$ should refer to the same `DState`. This might not be the case without copying the reference stored in *ht* locally, given that *ht* may be concurrently updated by resizing actions.

4.4 Resizing the hash table

Figure 6 describes the operations required to resize the hash table, i.e., to split each of the full buckets into two buckets, as well as to increase the size of the directory. We defer the discussion of bucket merging to Section 4.5.

Updating the directory. When an Insert operation fails because a bucket is full, the calling thread runs the `ResizeWF()` function (line 45). The algorithm executed by `ResizeWF()` follows the same basic principle as that executed by `ApplyWFOp()`. All threads willing to apply a resize action (here splitting a full bucket) will run two iterations of a loop where they will first make a local copy of the directory (line 119), then apply the required modifications on their local copy, and finally try to make their local copy the active state of the directory using a `CAS` instruction (line 124). However, there is no need to use bit vectors to determine the actions to be executed on the directory: all such actions will be resizing actions.

Making `ApplyWFOp()` and `ResizeWF()` wait free is not enough to ensure that Insert is wait free. We need to also ensure that a thread calls these functions a bounded number of times to complete an Insert. To this end, special care should be given to *pending resize actions* and pending update operations during resizing. A *pending resize action* is a resize action that must be performed in order for a pending update on a bucket to complete. Specifically, (i) a thread running a resize action has to execute all pending resize actions on the current state of the hash table, and (ii) when creating a new bucket, a thread has to execute all pending update operations on the newly created bucket. We explain the necessity of each of these two special actions in the next two paragraphs.

Consider the case where two threads, T_1 and T_2, need to execute `ResizeWF()` to complete an Insert operation each, on distinct buckets. Assume that T_1 needs to split bucket b_a and T_2 needs to split bucket b_c to complete their operations. If T_1 tries to update the directory in parallel with T_2, without executing the split of bucket b_c, there is a chance that only T_1 will manage to make its new computed state active using `CAS` (line 124), requiring T_2 to run again its resize action. The second time, a thread T_3 could be trying to split a bucket b_d in parallel with T_2 splitting b_c, and make

```
74  def SplitBucket(Bucket_p b): (Bucket_p, Bucket_p)
75      b0 ← new Bucket(b) # copy of bucket b
76      b0.depth ← b.depth + 1
77      b0.prefix ← b.prefix << 1 # bucket with prefix B1B2B3...0
78      b0.state ← new BState() # new empty BState
79      b0.state.results ← b.state.results #copy if the results array of b
80      b0.state.applied ← b0.toggle
81      b1 ← new Bucket(b0)
82      b1.prefix ← b0.prefix + 1 # bucket with prefix B1B2B3...1
83      for (k,v) in b.state.items: # insert the key-value pairs in new BStates
84          if Prefix(k, b0.depth) == b0.prefix:
85              b0.state.items[k] ← v
86          else:
87              b1.state.items[k] ← v
88      return b0, b1

90  def DirectoryUpdate(DState_p d, Bucket_p blist[]):
91      for b in blist:
92          if b.depth > d.depth: # doubling directory size is required
93              alloc d.dir with size 2^(d.depth+1)
94              copy all Bucket_p from previous dir to new d.dir
95              d.depth ← d.depth + 1
96          #compute the set of directory entries that should point to b
97          entries ← {e for e in 1..2^(d.depth) if Prefix(e, b.depth) = b.prefix}
98          for e in entries: # insert b in all required entries of d.dir
99              d.dir[e] ← b

101 def ApplyPendingResize(DState_p d, Bucket_p bFull):
102     for j in 1..n:
103         if Prefix(help[j].op.key, bFull.depth) == bFull.prefix:
104             if bFull.state.results[j].seqnum < help[j].seqnum:
105                 # help[j] is a pending request that was applying to bFull
106                 # bDest is the destination of help[j] in the current state
107                 bDest = d.dir[ Prefix(help[j].op.key, d.depth) ]
108                 while bDest is full:
109                     (b0,b1) ← SplitBucket(bDest)
110                     DirectoryUpdate(d, (b0, b1))
111                     #update bDest based on the new state of the directory
112                     bDest ← d.dir[ Prefix(help[j].op.key, d.depth) ]
113                 bDest.state.results[j].status ← ExecOnBucket(bDest, help[j])
114                 bDest.state.results[j].seqnum ← help[j].seqnum

116 def ResizeWF():
117     for k in 1..2:
118         oldD ← ht
119         newD ← new DState(oldD) # copy of object oldD
120         for j in 1..n: # iterates over operations in help
121             b ← newD.dir[ Prefix(help[j].op.key, newD.depth) ]
122             if b is full and b.state.results[j].seqnum < help[j].seqnum:
123                 ApplyPendingResize(newD, b)
124         CAS(ht, oldD, newD)
```

Figure 6: Code for the Resizable Hash Table.

T_2 fail again. As such a scenario could keep happening, a thread that executes `ResizeWF()` executes all pending resize actions in the current state of the directory in order to ensure wait-freedom.

Consider now the following scenario where multiple threads run Insert on a single bucket that is to be split. Assume that a thread T_1 applies an Insert on a full bucket b and so it calls `ResizeWF()`. At this time, it experiences some delay and in the meantime another thread T_2 executes resizing actions and replaces the full bucket b with two new buckets. Thread T_1 would now have to apply its Insert operation on one of the newly created buckets. However, suppose that other threads applied Insert operations in the meantime, so that this new bucket is full again. In this case, the operation of thread T_1 would fail again and T_1 would have to execute another resize action using `ResizeWF()`, with the risk that the same thing would happen again. To prevent such a scenario, a thread creating a new bucket executes all pending updates directed to this bucket.

Executing each operation exactly once. Since Insert operations may be executed through two different paths, namely during the execution of `ApplyWFOp()` (line 57) or during the execution of `ApplyPendingResize()` (line 113) which is called by `ResizeWF()` (line 123), we need a way to figure out when an operation has already been executed or is still pending. To this end, we use a per-thread sequence number *opSeqnum*. A thread T_i tags each update operation it executes with a distinct sequence number (line 40). When an operation of T_i has been successfully executed on a bucket b, its sequence number is copied into the entry i of the `results` array associated with $b.state$, i.e., `results[i].seqnum` is the sequence number of the last update executed by T_i on b.

The sequence number of the operations announced in `help[j]` (for some j) targeting a bucket b is compared to the corresponding entry `b.state.results[j].seqnum` to determine whether the operation is pending. This test is made both in `ApplyWFOp()` (line 56) and in `ApplyPendingResize()` (line 104) to decide which operations to execute on the bucket.

Note that in `ApplyWFOp()`, a thread cannot solely rely on the toggle vector of the bucket to decide which operations to execute (line 55). Indeed, since a thread starts an update operation by registering its operation in the `help` array (line 40), there is a chance that its operation would be executed during a resizing action even before it calls `ApplyWFOp()` and flips its bit in the toggle vector (line 50). As such, the only safe way to identify pending updates is using the sequence numbers. The toggle vector associated with each bucket is used to improve performance: It allows to identify very fast potentially pending updates without having to read the whole `help` array in `ApplyWFOp()`.

Detailed description of the resizing algorithm. We now describe the `ResizeWF()` function in detail. After making a local copy of the directory state, a thread iterates over the `help` array to find *pending resize actions* (lines 120-122); specifically, the thread looks in `help` for pending update operations that apply to a full bucket. For each pending resize action, the thread calls `ApplyPendingResize()` with the corresponding bucket, *bFull*, as parameter.

In `ApplyPendingResize()`, a thread should run all pending updates that are to be applied to bucket *bFull*. To this end, it iterates over the help array to find the pending operations that apply to *bFull* (lines 102-104). For such an operation, it selects the bucket *bDest* on which the operation should be applied in the current local directory state (line 107). As long as *bDest* is full, it splits *bDest* into two buckets $b0$ and $b1$ (call to `SplitBucket()` line 109), updates the directory with the two new buckets calling `DirectoryUpdate()` (line 110), and updates *bDest* based on the new state of the directory (line 112). Finally, it executes the pending operation on bucket *bDest* and updates the sequence number in $bDest.state.results$ accordingly (lines 113-114).

We remark that *bDest* can be different from *bFull* when executing line 107. The first operation applied on *bFull* will split *bFull* and replace it by non-full buckets. Hence, for other operations targeting *bFull*, *bDest* should be set to point directly to the appropriate newly created bucket. Note also that buckets might have to be split several times before completing a single INSERT operation (lines 108-112). The reason is that when one full bucket b is split into two new buckets $b0$ and $b1$, there is a chance that all items stored in b should be moved to just one of the two new buckets. For instance, in Figure 1b, when splitting bucket 010, all items will be stored in bucket 0100, which thus should be split again.

Function `SplitBucket()` takes, as a parameter, a reference to a bucket b of depth D with prefix "$B_1B_2 \dots B_D$" and returns references to two new buckets b_0 and b_1 of depth $D+1$, with prefixes "$B_1B_2 \dots B_D0$" and "$B_1B_2 \dots B_D1$". Key-value pairs stored in the `BState` of b are copied to the `BState` of $b0$ and $b1$ based on their prefix (lines 83-87). We recall that since b is full, the algorithm guaranties that its `BState` is immutable. Note also that $b.state.results$ is copied in $b0$'s and $b1$'s state (line 79) so that the results of operations previously applied to b are also available through $b0$ and $b1$. Finally, for each new bucket, its `applied` vector is initialized to be equal to its `toggle` vector (line 80): since all pending updates on the bucket are executed by the `ApplyPendingResize()` function, and since the bucket is not visible to other threads as long as the `CAS` instruction that updates the active directory has not been executed (line 124), its toggle vector should reflect no pending operations.

`DirectoryUpdate()` inserts in the directory the new buckets generated by `SplitBucket()`. The function doubles the size of the directory if needed, and stores the references of the new buckets in the appropriate entries of the directory. Although we do not detail all steps to be executed to double the size of the directory, the two main steps are: (i) allocating a new array corresponding to the new size of the directory (line 93), and (ii) copying all the bucket references from the previous array to the appropriate entries of the new array (line 94). Note that a new bucket might have to be inserted in more than one entry if the depth of the directory if bigger than the depth of the bucket (lines 97-99).

Avoiding losing updates. Our algorithm allows update operations on non-full buckets to run concurrently with resize actions. As pointed in Section 4.1, the two levels of indirection between the `DState` objects and the `BState` objects storing items ensure that updates are not lost in this case. Indeed, when a thread creates a new directory state during resizing, it copies the references of all existing buckets (line 119) in the new state. Since applying an update on a bucket b only modifies the `BState` reference stored in *b.state* (line 61), the newly created directory state will allow accessing the concurrently updated `BState` objects.

Note that since resizing requires full buckets to be immutable to avoid losing updates when running `SplitBucket()`, DELETE operations must not be executed on a full bucket (lines 64-65).

Compliance with the design rules. The described algorithm complies with the two design rules we have previously defined. (A) LOOKUP operations are executed without ever requiring any synchronization. The LOOKUP implementation is equivalent to a sequential implementation. (B) When no resize action is running, an update operation is executed by the instance of `PSim` of the corresponding bucket (function `ApplyWFOp()`) fully in parallel with operations applying to other buckets. The use of per-thread sequence numbers does not impair execution parallelism.

4.5 Merging buckets and shrinking the directory

We now provide a high level description of how our implementation copes with merging and shrinking. Merging buckets and shrinking the directory should both be run through the `ResizeWF()` function. Shrinking the directory can be implemented in the same way as doubling its size is done. Merging buckets is more complex. The basic idea is the same as for splitting: no update operations should be allowed to execute on buckets that can be merged (in order to avoid violating linearizability). However, there are two main differences between merging and splitting: (i) merging is applied to non-full buckets, and (ii) it involves more than one bucket.

To address the first point we use a mechanism to freeze a bucket, similarly to what is proposed in [19]. No update operations can be run on a frozen bucket even if it is not full. Instead, the thread willing to run an update operation on a frozen bucket should help running the merging action first. We have implemented freezing using a flag that is stored in the bucket state. The flag is modified by a thread using `ApplyWFOp()`. To address the second point, we perform merging in two steps: first, a thread tries to freeze all buckets involved in the merging, and then it calls `ResizeWF()` to perform the merging action.

Note that since the merging of buckets is done in several steps, it may fail: a thread may not manage to freeze all buckets involved in the merging action. The first reason why a thread may fail to freeze a bucket is that the bucket might be full. The second reason is that the bucket might be already frozen, e.g., because it is involved in another merging action. This might happen if two threads want to execute conflicting merging actions, for instance, one wants to create a bucket with prefix 001 and the other with prefix 0010. Since the two new buckets have a common prefix, the two merging actions involve overlapping sets of buckets. One of the two threads will not manage to freeze all required buckets for its action to take place. We remark that to avoid having both conflicting merge actions fail, all threads should freeze buckets in the same order. If a merge action fails, some buckets might have to be unfrozen. This operation is done during a directory update through the `ResizeWF()` function. It is determined by the user when merging will be triggered.

4.6 Corrrectness and Progress

When no resizing occurs, linearizability is proved following similar arguments as those used to prove `PSim` correct. When resizing occurs, we prove that: (1) each operation is executed exactly once, and (2) once an update operation has been executed, its modifications cannot be lost due to resizing. Due to lack of space, the full proof of correctness will be provided in the full version of the paper.

LOOKUP operations are obviously wait-free. Each INSERT operation calls at most two functions (`ApplyWFOp()` and `ResizeWF()`) that implement `PSim`. Since `PSim` is wait-free, it remains to show that functions called by these two functions execute a bounded number of instructions. `PSim` creates a copy of the state to manipulate. Hence, the code executed by any instance of `PSim` is sequential. Note that functions `ExecOnBucket()`, `SplitBucket()` and `DirectoryUpdate()` execute a bounded number of instructions. `ApplyWFOp()` calls `ExecOnBucket()` at most n times. So, the total number of executed instructions is bounded in this case.

The loops at lines 120 and 102 imply that the upper bound on the number of calls to ApplyPendingResize() by ResizeWF() is $O(n^2)$. The number of instructions run by ApplyPendingResize() is bounded by the number of times a bucket should be split in order to apply its pending operations. We conclude that the total number of instructions executed is bounded.

5 IMPLEMENTATION

The description provided in Section 4 assumes that a garbage collector (GC) is available. As pointed out in previous studies [2, 10], memory management can have a severe impact on the performance of concurrent data structures. So, we paid special attention to this issue while we were implementing our algorithm. Specifically, we use an epoch-based non-blocking garbage collector [9]. It is based on thread-local counters that are incremented when a thread performs an operation. The value of these counters are checked periodically to decide when some released memory can safely be reused. The frequency at which counters are checked depends on the size of the thread-local batches that are used to store references to elements that have been released: when a batch becomes full, the counters are checked to see if previous batches can be released.

Our implementation has to be able to efficiently allocate bucket states (BState) since any update operation requires the allocation of a new BState record. We implemented thread-local memory pools (heaps) that are optimized to allocate memory blocks of the size of a BState record. We also implemented a mechanism to allow memory that has been released by the GC to be inserted back in these thread-local heaps.

To further improve the performance of our algorithm in practice, we apply an optimization to reduce the size of BState objects. Namely, we reduce the size of the *results* array by storing the actual results of the operations in a separate shared array and storing in the BState an array of integers corresponding to indices in this shared array. To obtain the result of its executing operation, a thread i should read $results[i]$ in the BState, and then access the appropriate element of the shared array. This shared array is divided to non-overlapping blocks, one for each thread, so that each thread writes to its own block (with no need for synchronization).

We implemented the toggle vectors so that bits can be flipped efficiently using atomic Add (see [7]). Note also that when the CAS of lines 61 is successful, ApplyWFOp() can return immediately.

6 PERFORMANCE EVALUATION

6.1 Evaluated algorithms

We implemented our algorithm and the memory management component in C. The implementation allows activating or deactivating the use of local heaps for memory allocation. Our algorithm is called WF-Ext hereafter.

We compare the performance of WF-Ext with the performance of the lock-free implementations presented by Shalev and Shavit [21] (called LF-Split) and by Liu et al. [19] (called LF-Freeze). LF-Split is the reference implementation of the algorithm by Shalev and Shavit available online[4]. LF-Freeze is the C version of the code provided by the authors[5]. To ensure a fair comparison, we also created modified versions of these algorithms. A modified version is identified with the suffix -U. Specifically, for LF-Split, we removed the global counter, used for resizing decisions, that is accessed at the end of the execution of each update operation. Instead, resizing decisions are taken based on the size of individual buckets. For LF-Freeze, the original version of the authors only stored keys. We modified it to store key-value pairs. Also, in the original version, inserting a key that is already present implies no modification of the hash table. In our modified version, the value should be updated to comply with the semantics of the operations we described in Section 4. Note that this has a big impact on the performance results because with this semantic no Insert boils down to a LookUp (This explains the differences between our graphs and the ones presented in [19]). Starting from these modified versions, we also implemented a version of each algorithm that uses our memory management component (marked with suffix -M). LF-Freeze-M uses fixed-size buckets (instead of varying size buckets) to better leverage our local heaps. In its original implementation, LF-Split relies on the system memory allocator, while LF-Freeze relies on its own epoch-based GC and does not use local heaps.

Our evaluation also includes a simple *blocking* non-resizable hash table (called Lock) where each bucket is protected using a lock and each operation on the bucket has to acquire this lock.

6.2 Experimental setup

We have performed experiments in two multiprocessor architectures: a 48-core machine equipped with 4 Intel Haswell-EX E7-4830-v3 processors and a 64-core machine equipped with 4 Intel Broadwell E7-4850-v4 processors. All experiments showed similar performance behavior in both machines. Hence, we only present the results of our experiments on the larger and newer 64-core machine. This machine features 256GB of RAM distributed over 4 NUMA nodes. The operating system is Linux with kernel 4.9.9 and the compiler is gcc 5.4. All codes are compiled at the maximum level of optimization. All tests were run 8 times with a 5-second duration per test. We present the average throughput over all runs.

As discussed in other studies, the underlying memory allocator has a big performance impact. All evaluations were run with two allocators: the *glibc* allocator and the TCMalloc allocator[6]. For each experiment and each algorithm, we display results for the allocator that provides the best throughput. Information about the selected allocator is provided in the legend of the graphs: "G" for the *glibc* allocator and "T" for TCMalloc. We also test the two main NUMA policies for physical memory allocation, that is *local* (named "L" in graphs' legends) and *interleave* (named "I"), and display the best result. The NUMA policy is enforced using the *numactl* program.

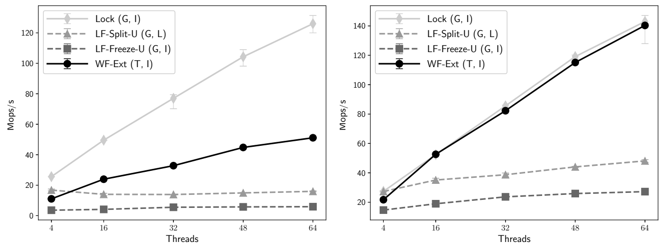
Several parameters can be configured for each algorithm, namely, the bucket size, the size of the batches used to store released memory records before running the Garbage Collector (GC), and the size of the thread-local heaps. We performed a wide range of experiments to identify parameters that suits all algorithms. We use these parameters throughout all experiments: buckets of size 8, GC batches of size 256, thread-local heaps of size 8.

Finally, in all experiments, the operations to be executed, as well as their parameter (the key), are randomly selected using the

[4] http://www.memoryhole.net/kyle/2011/06/02/

[5] https://github.com/mfs409/nonblocking/tree/master/tsx_acceleration/chash

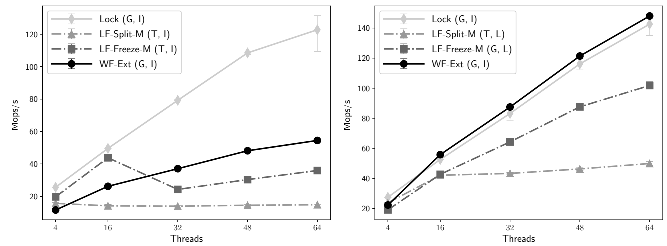
[6] http://goog-perftools.sourceforge.net/doc/tcmalloc.html



(a) 50% LookUp operations.

(b) 90% LookUp operations.

Figure 7: Directory-stable state throughput in the Intel 64-core machine using $1K$ items. No local heaps.



(a) 50% LookUp operations.

(b) 90% LookUp operations.

Figure 8: Directory-stable state throughput in the Intel 64-core machine, using $1K$ keys. With local heaps.

TinyMT[7] pseudo-random number generator. This generator produces numbers with a uniform distribution.

6.3 Directory-stable performance

First, we study the performance of hash tables in the most common scenario, that is, when resizing is rare. More specifically, we evaluate the throughput of the implementations starting from a hash table where we have already inserted key-value pairs corresponding to half of the keys manipulated during the experiment and we run workloads with the same amount of Delete and Insert operations.

Our first test uses $1K$ keys. Figures 7a and 7b correspond to loads with 50% and 90% of LookUp operations, respectively. The rest of the operations are equally divided between Insert and Delete. The figures compare the performance of our wait-free algorithm with the performance of the two modified lock-free algorithms LF-Split-U and LF-Freeze-U, when relying on the underlying system memory allocator (*glibc* malloc or TCMalloc) for memory allocation. We choose to display LF-Split-U because it constantly outperforms the original LF-Split algorithm (which confirms the importance of rule (B)). We choose to display LF-Freeze-U because a comparison with LF-Freeze would not be fair due to the differences in the implementation of Insert (see Section 6.1).

In Figure 7, our hash table outperforms the two lock-free algorithms. The high throughput of our algorithm is mostly due to highly efficient LookUpThis is confirmed by the large improvement observed when increasing the percentage of LookUps to 90%. We attribute the lower performance of the two lock-free algorithms to different factors. Specifically, for LF-Split-U, the reason is twofold: (i) the algorithm does not comply with design rule (A), implying less efficient LookUp operations in this case; (ii) the buckets are implemented as lists of items, which makes item search less efficient than with buckets implemented as arrays. In the case of LF-Freeze-U the main reason we identified for the lower performance is that the bucket size is not fixed. It is adapted to the number of items to store. Since a new bucket should be allocated for each update operation, it creates an unfriendly workload for the memory allocation system.

Figure 8 presents the same experiment as in Figure 7, this time using both our GC and the local heaps for all non-blocking algorithms (-M suffix). This figure shows that even when compared using the exact same memory management component, WF-Ext is more efficient than the lock-free algorithms. In this experiment, with 90% LookUps, our algorithm is up to 47% better than the second best non-blocking algorithm. Still, the performance of LF-Freeze-M shows that using our memory management component and fixed-size buckets greatly improves the performance of LF-Freeze. On the other hand, LF-Split does not benefit from using local heaps.

To assess the performance of our algorithm when storing a larger number of items, Figure 9 presents the same experiment as in Figure 8, this time when keys are selected among $256K$ keys. For the sake of completeness, we present the performance of two versions of each lock-free algorithm. For LF-Split, we present the original version and LF-Split-U (LF-Split-M is not considered because it does not reach the performance of LF-Split-U). For the other algorithm, we present both LF-Freeze-U and LF-Freeze-M.

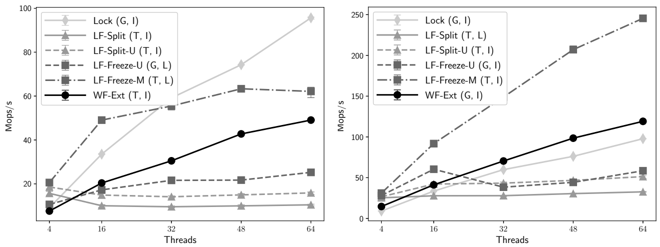
Compared to the algorithms described in the related work (original algorithms or -U versions), the performance of our algorithm is again by far the best. It even outperforms the lock-based hash-table when the percentage of LookUps is high. The high performance of LF-Freeze-M demonstrates that the algorithm proposed by Liu et al. can benefit from the modifications we suggest to efficiently manage memory allocation. Once memory management issues are solved, LF-Freeze-M outperforms WF-Ext because its update operations are less complex since it implements a weaker progress condition.

The fact that WF-Ext outperforms all lock-free implementations when the hash table has a small number of buckets can be surprising, especially since existing wait-free resizable hash tables are performing much worse than their lock-free counterparts [19]. This result is due to the use of the PSim universal construction [7]. Indeed, PSim performs especially well on contented objects since it employs the technique of *combining* the operations of different threads that are applied to the same object [12].

6.4 Resizing efficiency

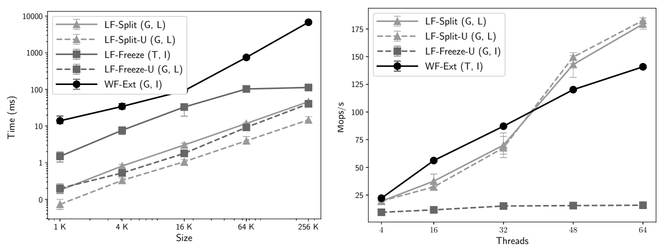
To evaluate resizing performance, we run a test where the hash table starts with only 2 buckets. Multiple threads start inserting items randomly, and we measure the time it takes for the hash table to reach its final size. To have a more realistic workload, threads also execute LookUp operations on the items with a 50% probability. Figure 10a presents the results. Note that to make the figure readable, we use log scales on both axis.

[7] https://github.com/MersenneTwister-Lab/TinyMT



(a) 50% LookUp operations. **(b) 90% LookUp operations.**

Figure 9: Directory-stable state throughput in the Intel 64-core machine, using $256K$ keys.



(a) Resizing efficiency **(b) Throughput of 5-second runs with 1k items**

Figure 10: Performance with Resizing. Tests with 64 threads

The figure shows that the resizing performance of our algorithm is much lower than that of evaluated lock-free algorithms. However, the experiment of Figure 10b illustrates that WF-Ext still performs well under resizing, if the cost of resizing is amortized over long runs. In this experiment, we evaluate the throughput on a 5-second run when starting from a hash table with 2 buckets and manipulating $1K$ items in a load with 90% LookUp and 10% Insert operations. In this case, WF-Ext reaches the same throughput as when resizing actions are rare (Figure 8b). Thus, in sufficiently long runs the performance impact of resizing is not significant. We note that LF-Split achieves much better performance than in previous experiments. Indeed, in this test, LookUps for LF-Split are more efficient since no item is ever deleted, confirming the validity of design rule (A).

7 CONCLUSION

This paper presents a resizable wait-free hash table based on extendible hashing. The design of our algorithm is based on two design rules that aim at preserving the natural parallelism of concurrent hashing in the most common case, that is, when no resizing action occurs. Leveraging several instances of the PSim universal construction to ensure wait-freedom, our implementation achieves unprecedented performance for a wait-free resizable hash table. More generally, our approach provides a new trade-off in the performance of resizable hash tables. Namely, experiments run on large multicore architectures, show that, at the cost of more expensive resizing actions, our algorithm largely outperforms lock-free hash tables described in related studies when resizing actions are rare, while providing a stronger progress guarantee.

ACKNOWLEDGMENTS

Part of this work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671578. The authors would like to thank Darko Petrović and André Schiper for following the initial stages of this work and providing feedback.

REFERENCES

[1] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, 2010.

[2] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 631–644, 2015.

[3] C. S. Ellis. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2Nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 106–116, 1983.

[4] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2), July 1988.

[5] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, Sept. 1979.

[6] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[7] P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, pages 1–46, 2013.

[8] S. Feldman, P. LaBorde, and D. Dechev. Concurrent multi-level arrays: Wait-free extensible hash maps. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 155–163, 2013.

[9] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[10] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, 2015.

[11] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, PODC '02, 2002.

[12] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[15] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[16] L. Jenkins, T. Zhou, and M. Spear. Redesigning Go's Built-In Map to Support Concurrent Operations. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[17] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, 2012.

[18] D. Lea. Package util.concurrent release 1.3.4 (2003). http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[19] Y. Liu, K. Zhang, and M. Spear. Dynamic-sized Nonblocking Hash Tables. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, 2014.

[20] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, 2002.

[21] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006.

[22] N. Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, Mar. 2011.

[23] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference*, 2011.