Towards an Elastic Lock-Free Hash Trie Design

Miguel Areias and Ricardo Rocha
CRACS & INESC TEC, Faculty of Sciences, University of Porto, Portugal
Email: {miguel-areias,ricroc}@dcc.fc.up.pt

***Abstract*—A key aspect of any hash map design is the problem of dynamically resizing it in order to deal with hash collisions. In this context, *elasticity* refers to the ability to automatically resize the internal data structures that support the hash map operations in order to meet varying workloads, thus optimizing the overall memory consumption of the hash map. This work extends a previous lock-free hash trie design to support elastic hashing, i.e., expand saturated hash levels and compress unused hash levels, such that, at each point in time, the number of levels in a path matches the current demand as closely as possible. Experimental results show that elasticity effectively improves the search operation and, in doing so, our design becomes very competitive when compared to other state-of-the-art designs implemented in Java.**

I. Introduction

Hash maps are a very common and efficient data structure used to store data that can be organized as (K, C) pairs, where the mapping between the unique key K and the associated content C is given by a hash function. Hash tries (or hash array mapped tries) are a tree-based data structure with nearly ideal characteristics for the implementation of hash maps [1]. A key aspect of any hash map design is the problem of dynamically resizing it in order to deal with hash collisions. This includes increasing the size of the underlying data structure and remapping (or rehashing) all the existing keys to new locations and decreasing (or compressing) the size of the data structure when a certain amount of keys are removed.

An alternative to all-at-once rehashing is to perform resizing gradually or incrementally, thus affecting just a small part of the entire data structure. The advantages of compressing tree-based data structures are well-known in the literature [2], [3]. Compression can be done at shallow or deeper trie levels, but a key advantage is that it can be done concurrently with the other operations. Two good examples are: (i) the B*-tree proposal [4], which supports a compression procedure that runs concurrently with regular operations, such as searches, insertions and removals, to merge nodes that are underfull; and (ii) the relaxed B-slack trees proposal [5] that supports a similar concurrent absorb operation that reduces the number of levels in the data structure.

In this context, *elasticity* refers to the ability to automatically resize the internal data structures that support the hash map operations in order to meet varying (local) workloads, thus optimizing the overall memory consumption of the hash map. Tree-based hash maps are $\mathcal{O}(\log_E K)$, where E represents the branching factor in a hash level and K is the overall number of keys inserted in the hash map. Elasticity will work on adjusting the depth of the internal hash levels within a hash map to the number of keys K that the hasp map holds at any given instant of the execution. Thus, elasticity reduces, not only, memory consumption, but can also potentially reduce the execution time, since the number of levels to be traversed when trying to operate a key is expected to be lower.

In this work, we propose a novel concurrent hash trie design that puts together the following characteristics: (i) use fixed size data structures; (ii) use persistent memory references; (iii) be lock-free; (iv) store sorted keys; and (v) be elastic. In previous work [6], Areias and Rocha proposed a concurrent hash trie design that supports most of the characteristics above with the exception of elasticity. This work extends that previous design to also support *elastic hashing*, i.e., expand saturated hash levels and compress unused hash levels, such that, at each point in time, the number of levels in a path matches the current demand as closely as possible. To the best of our knowledge, none of the available alternatives in the literature fulfills all the above five characteristics simultaneously.

The remainder of the paper is organized as follows. First, we introduce relevant background and present the main ideas of our design. Next, we describe in detail the key algorithms required to easily reproduce our implementation. Then, we present a set of experiments comparing our design against other state-of-the-art concurrent hash map designs. At the end, we present conclusions and further work directions.

II. Background

The first correct CAS-based lock-free list-based set design was introduced by Harris [7]. Later, Michael improved Harris work by presenting a design that was compatible with all lock-free memory management methods and Michael used this design as the building block for lock-free hash maps [8]. Skip lists is an alternative and more efficient data structure to plain linked lists that allows logarithmic time searching, insertions and removals by maintaining multiple hierarchical layers of linked lists where each higher layer acts as an *express lane* for the layers below. Concurrent non-blocking skip lists were later implemented by Herlihy *et al.* [9] and Shalev and Shavit [10].

An essential property of the trie data structure is that common prefixes are stored only once [11], which in the context of hash maps allows us to efficiently solve the problems of setting the size of the initial hash map and of dynamically resizing it in order to deal with hash collisions. Prokopec *et al.* presented the CTries [12], a non-blocking concurrent hash trie based on shared-memory single-word CAS instructions. The CTries introduce a non-blocking, atomic constant-time *snapshot operation*, which can be used to implement operations requiring a consistent view of a data structure at a single point in time.

More recently, Areias and Rocha presented a novel lock-free hash trie design that combines hashing with sort and tree search algorithms to support additional important properties, such fixed-size data structures, persistent references and sorted keys [6]. Internally, the design has only two types of data structures, *hash arrays of buckets* and *leaf nodes*. The leaf nodes store the key/content pairs and the hash arrays of buckets implement a hierarchy of hash levels of fixed size 2^w. To map a key K into this hierarchy, it first computes the hash value h for K and then uses chunks of w bits from h to index the entry in the appropriate hash level, i.e., for each hash level H_i, it uses the i^{th} group of w bits of h to index the entry in the appropriate bucket array of H_i. To deal with collisions, the leaf nodes form a linked list in the respective bucket entry until a threshold is met and, in such case, it executes an expansion operation to update the nodes in the linked list to a new hash level H_{i+1}. This hierarchical organization is also the basis for the new design that we present next.

III. Our Design By Example

In this section, we focus the discussion on our design for elastic hashing, namely, on how the insert, expand and remove operations can work concurrently in a lock-free fashion with the new compress operation.

A. Inserting Keys and Expanding Hash Levels

We begin with Fig. 1 showing a small example that illustrates how the concurrent insertion of nodes is done in a hash level. Figure 1(a) shows the initial configuration for a hash level H_i. Each hash level is formed by a bucket array of 2^w entries and by a header, which includes a backward reference to the previous hash level, a hash level identifier and a key representative of the hash level, respectively, values P_i, i and K_1 in Fig. 1 (in Fig. 1(a), the key representative is marked as '–' since the hash level is still empty). For the root level, the backward reference is nil.

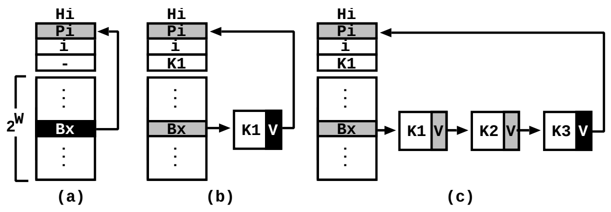

Fig. 1: Insert operation in a hash level

The bucket entries are initialized with a reference to the current hash level. In Fig. 1(a), B_x represents a particular bucket entry of H_i. Each bucket entry stores either a reference to a hash level or a reference to a separate chaining mechanism, using a chain of leaf nodes, that deals with the hash collisions for that entry. Each leaf node includes a tuple that holds both a reference to a next-on-chain leaf node and the state of the node, which can be valid (V) or invalid (I). The initial state of a node is valid. Figure 1(b) shows the configuration after the insertion of node K_1, on the bucket entry B_x, and Fig. 1(c) shows the configuration after the insertion of nodes K_2 and K_3, also in B_x. The insertion of nodes is done at the end of the chain and a new inserted node closes the chain by referencing back the current hash level.

When the number of valid nodes in a chain exceeds a threshold value, then the corresponding bucket entry is expanded with a new hash level and the nodes in the chain are remapped in the new level. To keep keys sorted, a *xor* operation is applied between the hash values of the key being inserted and the key representative of the hash level, to check in which chunk of bits they first differ. If they differ in a higher chunk of bits than the hash level chunk identifier, then a new hash level is inserted in a deeper level (this is called *front-expansion*). Otherwise, a new hash level is inserted in a shallow level (this is called *back-expansion*) [6].

We next describe the front-expansion operation in more detail. Starting from the configuration in Fig. 1(c), Fig. 2 illustrates the front-expansion operation to a second level hash for the bucket entry B_x. The front-expansion operation is activated whenever a thread T trying to insert a key K meets the following two conditions: (i) K is not found in the current chain of leaf nodes, and (ii) the number of valid nodes in the chain observed by T is equal to the threshold value corresponding to the number of collisions allowed (in what follows, we consider a threshold value of three keys). In such case, T starts by pre-allocating a second level hash H_k, with all entries referring the respective level and with a key representative consisting of the key in the chain (K_2 in the example of Fig. 2) that differs in the lower chunk of bits from the new key that is being inserted by T.

The new hash level H_k is then used to implement a synchronization point with the current insertion point (node K_3 in Fig. 2(a)) that will correspond to a successful CAS operation trying to update H_i to H_k (Fig. 2(b)). From this point on, the insertion of new nodes on B_x will be done starting from the new hash level H_k. If the CAS operation fails, that means that another thread has gained access to K_3 and, in such case, T aborts its front-expansion operation. Otherwise, T starts the remapping process of placing the valid nodes K_1, K_2 and K_3 in the correct bucket entries in the new level. Figures 2(c) to 2(f) show the remapping sequence in detail. For simplicity of illustration, we will consider only the entries B_y and B_z on level H_k and assume that K_1, K_2 and K_3 will be remapped to these bucket entries.

In order to ensure lock-free synchronization, at any time, any thread must be able to read all the available nodes and insert/remove nodes without any delay from the remapping process. To guarantee both properties, the remapping process is thus done in reverse order, starting from the last node on the chain, initially K_3. Fig. 2(c) shows the hash trie configuration after the successful CAS operation that adjusted node K_3 to entry B_z. After this step, B_z is no longer an insertion point (gray background) and K_3 becomes the next insertion point (black background) for the insertion of new nodes on B_z. Note

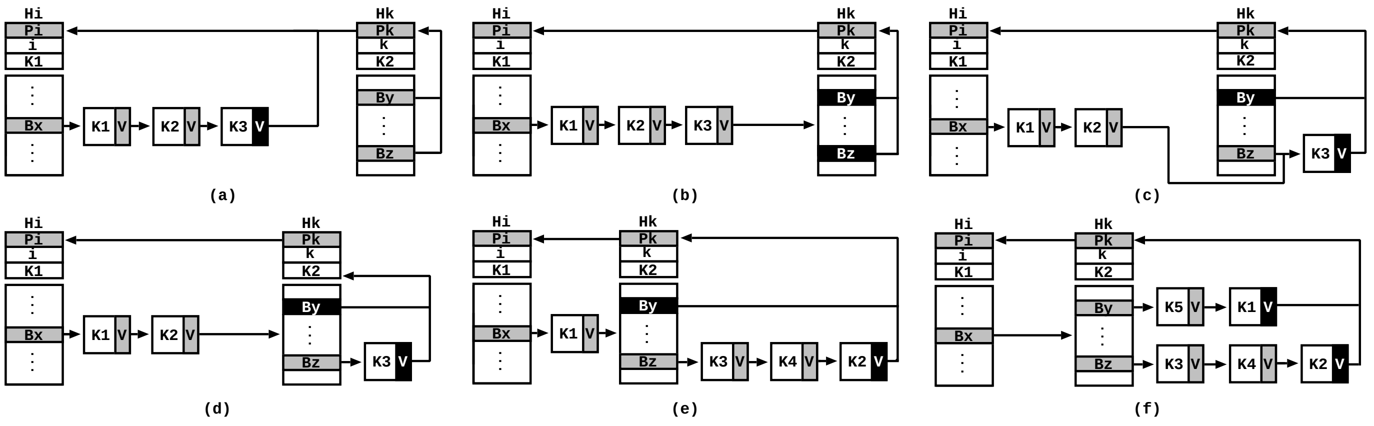
Fig. 2: Front-expansion with the concurrent insertion of nodes

that the initial chain for B_x has not been affected yet, since K_2 still refers to K_3. Next, on Fig. 2(d), the chain is adjusted and K_2 is updated to refer to the second level hash H_k. The process then repeats for K_2 (the new last node on the chain for B_x). First, K_2 is remapped to entry B_z and then it is removed from the original chain, meaning that the previous node K_1 is updated to refer to H_k (Fig. 2(e)). Finally, the same idea applies to node K_1. In the continuation, K_1 is also remapped to a bucket entry on H_k (B_y in the figure) and then removed from the original chain, meaning in this case that the bucket entry B_x itself becomes a reference to the second level hash H_k (Fig. 2(f)). Concurrently with the remapping process, other threads can be inserting nodes in the same bucket entries for the new level. This is shown in Fig. 2(e), where a node K_4 is inserted before K_2 in B_z and in Fig. 2(f), where a node K_5 is inserted before K_1 in B_y.

B. Removing Keys and Compressing Hash Levels

In this subsection, we begin by describing how the concurrent removal of nodes is done in a hash level and how it triggers the compress operation. Then, we show how the compress operation is done in a lock-free fashion.

A remove operation can be seen as a sequence of two steps: (i) the *invalidation step*; and (ii) the *unreachability step*. The invalidation step searches for the node N holding the key to be removed and updates the node state from valid to invalid. The unreachability step then searches for the valid data structures B and A, respectively before and after N in the chain of nodes, in order to bypass node N by chaining B to A. Starting again from the configuration in Fig. 1(c), where all keys are valid, Fig. 3 illustrates how the concurrent removal of nodes is done.

Consider that a thread T wants to remove the key K_2. T begins the invalidation step by searching for node K_2 and by marking it as invalid (Fig. 3(a)). In the continuation, T searches for the valid data structures before and after K_2, nodes K_1 and K_3 in this case. The next step is shown in Fig. 3(b), where node K_1 is chained to node K_3, thus bypassing node K_2. From this point forward, node K_2 is unreachable from the chain (unreachability step). The reader

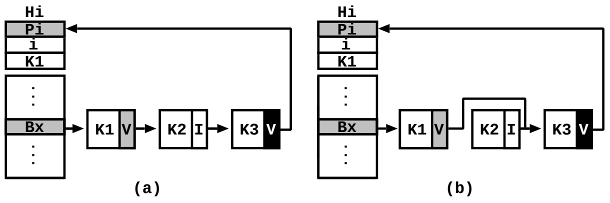
Fig. 3: Remove operation in a hash level

can observe that, the chaining references of unreachable nodes are left in a consistent state, allowing all late threads reading those nodes to be able to recover to a valid data structure.

The removal of a key might trigger the compression of the (leaf) hash level H_i where the key has found, if all bucket entries of H_i are found empty. To keep the lock-freedom property, the compress operation relies on a new special node, named *compression node*, used to mark an undergoing compress operation and in two key procedures: (i) the *freezing* procedure, used to mark all bucket entries as ready for compression; and (ii) the *unfreezing* procedure, used to abort an unsuccessful compression. At the implementation level, the compress operation: (i) does not keep track of which hash levels are being traversed by a thread; (ii) does not keep track of the number of buckets that are empty on a hash level; and (iii) does not use snapshots to compress the hash levels, because keys are stored in chain nodes and not in the hash buckets. Figure 4 illustrates an example of a successful compress operation.

Figure 4(a) shows the initial configuration of the hash levels, where bucket entry B_x is referring to the hash level H_k, which has only one node (with the key K_2) in the bucket entry B_z (all the remaining bucket entries are empty). The compress operation will then be triggered when a thread T_1 removes the key K_2 and becomes aware that B_z is empty (Fig. 4(b)). T_1 then uses the key representative K_2 of H_k to find the corresponding bucket entry B_x in the previous hash level H_i in order to insert the special compression node F, meaning that a freezing procedure is undergoing in the hash level H_k

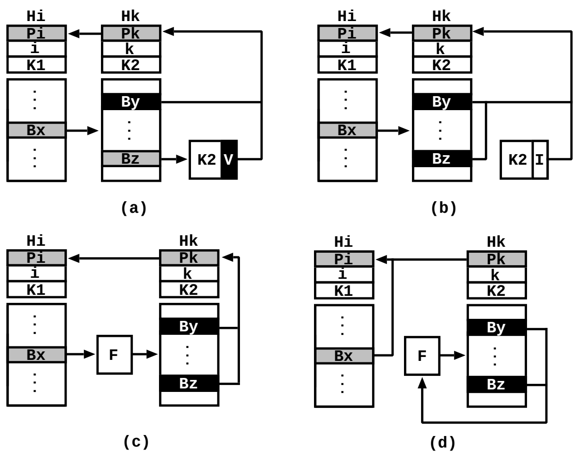

Fig. 4: A successful compress operation

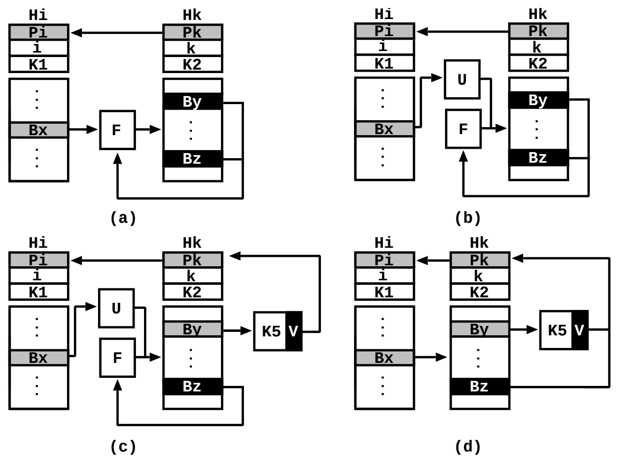

Fig. 5: Aborting an undergoing compress operation

(Fig. 4(c)). After the insertion of F, T_1 traverses all bucket entries in H_k and, for each bucket entry, it applies a CAS operation trying to update the entry's reference to node F. If one of the bucket entries is not empty (i.e., the CAS has failed), then T_1 aborts the freezing procedure and starts the unfreezing procedure. Otherwise, the freezing procedure has succeeded and all bucket entries are referring node F, in which case T_1 applies a final CAS on B_x to remove the hash level H_k from the data structure (Fig. 4(d)).

Note that, while a thread is trying to compress a hash level, other threads can be searching, removing or inserting keys in the hash level under compression. Whilst the search and remove operations cannot collide with the compress operation, the insert operation can. For instance, consider again the example in Fig. 4 and assume that a second thread T_2 is preempted in H_k, at the time of the configuration in Fig. 4(c). Later, if T_2 is resumed after the configuration in Fig. 4(d), then it must be able to detect that H_k has been compressed (and is not valid anymore) and must be able to position itself in a valid hash level. Otherwise, if T_2 is resumed before the configuration in Fig. 4(d), it must somehow synchronize with T_1 in order to be able to complete its insertion operation. In both situations, T_2 knows about the existence of a compress operation when it reaches the compression node F (note that F can also be reached from the bucket entry B_x in H_i but, in such case, the traversal can continue as usual to H_k). By rereading the reference in B_x, T_2 can check if the compression is undergoing (case in which B_x still refers F) or has already completed. If the compression is undergoing, then T_2 notifies T_1 to abort the compression before proceeding with its insert operation. Figure 5 illustrates the situation where a thread T_1 is compressing the hash level H_k and a thread T_2 wants to insert a key K_5.

Figure 5(a) shows the initial configuration where T_1 has already updated all bucket entries from H_k to refer F and is about to complete the process with the final CAS on B_x. Now consider that, due to preemption, T_1 suspends before updating B_x, and that, in the meantime, T_2 is trying to insert K_5 on B_y. T_2 follows the reference in B_y and reaches node F (thus knowing about the existence of an undergoing compress operation). T_2 then needs to notify T_1 to abort the compression, and for that it replaces F with another special compression node U, meaning that an unfreezing procedure needs to be done (Fig. 5(b)). Once this notification succeeds, T_2 can proceed by inserting K_5 in B_y (Fig. 5(c)). It is important to notice that if another thread T_3 was also trying to abort the compression (e.g., to insert another key), it would not have to insert a second node U, since one is enough to trigger the unfreezing procedure. Later, when T_1 resumes and tries to apply the CAS on B_x, the CAS will fail. In the continuation, T_1 will notice the existence of node U and will start the unfreezing procedure. This requires traversing again all bucket entries in H_k to unfreeze them. At the end, T_1 applies a CAS operation on B_x to remove U. Figure 5(d) shows the corresponding final configuration.

IV. Algorithms

This section presents the most relevant algorithms of our design.

A. *SearchRemoveKey*

Algorithm 1 shows the pseudo-code for the search/remove operation of a given key K in a given hash level H.

The algorithm begins by getting the bucket entry B from H that fits the key K and by reading the reference R in B (lines 1–2). Next, the algorithm checks if R is a reference to a compression node (lines 3–4), case in which R is updated by following the chain, thus making R necessarily a reference to a hash level (to H or to a second hash level). In the continuation, the algorithm then checks if R is a reference to a hash level (lines 5–9), case in which it simply returns if the current chain is empty (line 7) or restarts if R references a second hash level (line 9).

Otherwise, R holds a reference to a chain node and the algorithm traverses the chain of nodes looking for a valid node holding K. If a valid chain node holding K is found, the algorithm proceeds to remove it (lines 11–13). Otherwise, the

Algorithm 1 *SearchRemoveKey(Key K, Hash H)*

```
1: $B \leftarrow GetBucket(K, H)$
2: $R \leftarrow NextRef(B)$
3: if $IsCompressionNode(R)$ then
4:    $R \leftarrow NextRef(R)$
5: if $IsHashLevel(R)$ then
6:    if $R = H$ then  // empty chain, K is not in H
7:       return
8:    else  // R references a second hash level
9:       return $SearchRemoveKey(K, R)$
10: repeat  // traverse the chain of nodes
11:    if $IsValidChainNode(R) \wedge Key(R) = K$ then  // key found
12:       if $MakeChainNodeInvalid(R)$ then
13:          return $MakeChainNodeUnreachable(R, H)$
14:    $R \leftarrow NextRef(R)$
15: until $IsHashLevel(R)$
16: if $R = H$ then  // chain ended in the same hash level
17:    return
18: $R \leftarrow GetHashLevel(R, Level(H) + 1)$
19: return $SearchRemoveKey(K, R)$
```

chain of nodes was traversed and K was not found, which means that R holds now a reference to a hash level. If R holds a reference to H then no expansion/compression operation has interfered with the search of K, thus the algorithm can simply return (line 16–17). Otherwise, R holds a reference to a deeper hash level, thus the algorithm restarts in the hash level after H (lines 18–19).

B. *MakeChainNodeUnreachable*

Algorithm 2 presents next the pseudo-code for turning unreachable a given node N in a given hash level H. Remember that, in the unreachability step, we need to search for the valid data structures BR (*before reference*) and AR (*after reference*), respectively before and after N in the chain of nodes, in order to bypass node N by chaining BR to AR.

The algorithm begins by setting R and AR with the next valid data structure starting from N (lines 1–2). If R is a chain node, then R is updated with the hash level at the end of the chain (lines 3–4). Otherwise, R already refers a hash level. In both cases, at the beginning of line 5, R refers a hash level. If R refers a deeper hash level, the process is restarted in the hash level after H (lines 25–26). Otherwise, the algorithm ended in the same hash level H (lines 5–24) and it proceeds to compute the valid data structure BR before N. For that, it starts from the bucket entry B in H that fits the key on N and traverses the chain of nodes looking for the following valid data structures until reaching N or a hash level (lines 6–12).

At the end of the traversal, if R reaches N then we are in condition to bypass N by chaining BR to AR and thus make N unreachable (lines 13–22). For that, the algorithm applies a CAS operation to BR trying to update it from the reference saved in BRN to AR and keeping the node state as valid if BR is a chain node (line 20). However, if BR refers a bucket entry and AR refers to H (lines 14–18), then the algorithm tries to compress the hash level H (more details later). Notice

Algorithm 2 *MakeChainNodeUnreachable(Node N, Hash H)*

```
1: $R \leftarrow GetNextHashLevelOrValidChainNode(N)$
2: $AR \leftarrow R$
3: if $IsChainNode(R)$ then
4:    $R \leftarrow GetNextHashLevel(R)$
5: if $R = H$ then  // chain ended in the same hash level
6:    $B \leftarrow GetBucket(Key(N), H)$
7:    $R \leftarrow B$
8:    repeat
9:       $BR \leftarrow R$
10:      $BRN \leftarrow NextRef(BR)$
11:      $R \leftarrow GetNextHashLevelOrValidChainNodeOrN(R, N)$
12:   until $R = N \vee IsHashLevel(R)$
13:   if $R = N$ then  // we are in condition to bypass N
14:      if $BR = B$ then  // no valid chain nodes found
15:         if CAS$(NextRef(BR), BRN, AR)$ then
16:            if $AR = H$ then  // try to compress H
17:               $CompressHashLevel(Key(N), H)$
18:            return
19:      else
20:         if CAS$(Next(BR), (BRN, valid), (AR, valid))$ then
21:            return
22:      return $MakeChainNodeUnreachable(N, H)$
23:   if $R = H$ then  // N is already unreachable
24:      return
25: $R \leftarrow GetHashLevel(R, Level(H) + 1)$
26: return $MakeChainNodeUnreachable(N, R)$
```

that if the CAS operation fails, it means that the reference in BR has changed somewhere between the instant where it was found valid and the CAS execution. In such case, the process is restarted (line 22), thus forcing the algorithm to converge to a configuration where all invalid nodes are made unreachable.

Otherwise, if R ends in a hash level at the end of the traversal, that means that N is not on H. Therefore, if R refers to H that means that N is already unreachable, thus the algorithm simply returns (lines 23–24). Otherwise, R refers a deeper hash level and the process is restarted in the hash level after H (lines 25–26).

C. *CompressHashLevel*

Finally, Alg. 3 presents the pseudo-code for trying to compress a hash level. The algorithm receives as arguments the key K that triggered the compress operation and the hash level H to be compressed.

The algorithm begins by getting the previous hash level PH and if it does not exist, it means that the algorithm is trying to compress the root hash level, thus it returns (lines 1–3). Otherwise, it gets the bucket entry B from PH that fits K, allocates a freezing compression node F (with the next reference to H), and applies a CAS on B in order to insert F and thus mark the beginning of the freezing procedure (lines 4–6). If the CAS fails, then B is not referring to H anymore, thus the algorithm simply returns (line 28).

The freezing procedure starts by calling *FreezeHashLevel()* to freeze the bucket entries in the hash level H (line 7). If it fails, the algorithm then allocates an unfreezing compression

Algorithm 3 *CompressHashLevel(Key K, Hash H)*

```
1:  $PH \leftarrow PrevHashLevel(H)$
2:  if $PH = nil$ then  // abort if trying to compress the root hash level
3:      return
4:  $B \leftarrow GetBucket(K, PH)$
5:  $F \leftarrow AllocCompressionNode(H, freeze)$
6:  if CAS$(NextRef(B), H, F)$ then
7:      if $FreezeHashLevel(F, H)$ then
8:          repeat
9:              $R \leftarrow NextRef(B)$
10:             while $IsHashLevel(R)$ do  // back-expansion in the meantime
11:                 $PH \leftarrow R$
12:                 $B \leftarrow GetBucket(K, PH)$
13:                 $R \leftarrow NextRef(B)$
14:             if CAS$(NextRef(B), F, PH)$ then  // try to remove H
15:                 return $CompressHashLevel(K, PH)$
16:         until $IsCompressionNode(R, unfreeze)$
17:     else  // freezing failed
18:         $U \leftarrow AllocCompressionNode(H, unfreeze)$
19:         CAS$(NextRef(B), F, U)$
20:     $UnfreezeHashLevel(F, H)$
21:     repeat  // remove unfreezing node and restore configuration
22:         $R \leftarrow NextRef(B)$
23:         while $IsHashLevel(R)$ do  // back-expansion in the meantime
24:             $B \leftarrow GetBucket(K, R)$
25:             $R \leftarrow NextRef(B)$
26:         $U \leftarrow R$ // reached unfreezing compression node
27:     until CAS$(NextRef(B), U, H)$
28: return
```

node U to replace F and thus mark the beginning of the unfreezing procedure (lines 18–19). The unfreezing procedure follows on lines 20–27. If *FreezeHashLevel()* succeeds, the algorithm then updates the current previous hash level PH, in case any back-expansion has occurred in the meantime (lines 9–13), in order to apply the CAS that will remove H and thus effectively compress the data structure (line 14). In case of success, the algorithm then tries to recursively compress the previous hash level PH (line 15). Otherwise, in case of CAS failure, it means that another back-expansion occurred in the meantime or that an unfreezing compression node has been inserted by another thread. In the first case, we repeat the process of updating the previous hash level PH. Otherwise, we move to the unfreezing procedure.

The unfreezing procedure starts by calling *UnfreezeHash-Level()* to unfreeze the bucket entries in the hash level H (line 20). As before, the algorithm then repeats the process of finding the current previous hash level, in case any back-expansion has occurred in the meantime (lines 22–25), in order to reach the unfreezing compression node U (line 26) and apply the CAS that will restore the initial configuration and thus keep H in the data structure (line 27).

V. Performance Analysis

This section presents experimental results comparing our design with other state-of-the-art concurrent hash map designs. The environment for our experiments was a SMP system based in a NUMA architecture with 32-Core AMD Opteron (TM) Processor 6274 (2 sockets with 16 cores each) with 32GB of main memory, running the Linux kernel 3.18.x86_64 with OpenJDK's jdk-13.0.1. Although our design is platform independent, we have chosen to make its first implementation in Java, mainly for two reasons: (i) rely on Java's garbage collector to reclaim unreachable data structures; and (ii) easy comparison against other hash map designs. Some of the best-known hash map designs currently available are implemented in Java, such as the *Concurrent Skip-Lists (CSL)* from the Java's concurrency package and the *CTries (CT)* design from Prokopec *et al.* [12].

For our design, which we named *Free Fixed Persistent Hash Map (FFP)*, we used two versions: one with the support for elastic hashing disabled (*FFPS*) and a second one with elastic hashing enabled (*FFPE*). We ran both *FFPS* and *FFPE* designs with 8 buckets entries per hash level, a threshold value of 2 chain nodes for the hash collisions, and implementing sorted keys. To put all designs in perspective, Table I shows how the 4 designs support/implement the following characteristics: (i) be lock-free; (ii) use fixed size data structures; (iii) maintain the access to all internal data structures as persistent memory references; (iv) store sorted keys; and (v) be elastic.

TABLE I: Features support

Features / Designs	**CSL**	**CT**	**FFPS**	**FFPE**
Lock-freedom	✓	✓	✓	✓
Fixed size structures	-	✗	✓	✓
Persistent references	✓	✗	✓	✓
Sorted keys	✓	✗	✓	✓
Elastic hashing	-	✓	✗	✓

For the experiments, we used an open source benchmarking tool that allows to define benchmark sets with randomized operations, where each set has a pre-defined ratio of the most used operations in hash maps: (i) insertion of items; (ii) searching for items; and (iii) removal of items.

To spread threads among a set S of randomized operations, the tool equally divides S by the number T of running threads in such a way that each thread runs $\frac{S}{T}$ operations. To support non concurrent randomicity on each thread, we used JVM's *ThreadLocalRandom*. Additionally, we configured the benchmarking tool to run an initial setup, where some (or all) keys in the set $I = \{0, ..., 8^8 - 1\}$ are pre-inserted in the hash map design, and then we measure the execution time of running 1, 4, 8, 16, 24 and 32 threads, with 8^8 (16,777,216) random operations for random keys in set I. To measure the execution times, we ran each benchmark 5 times beforehand to warm-up the JVM, and then we took the average execution time of the next 10 runs.

A. Elasticity Overheads

We begin the experimental results with a comparison between the *FFPS* and *FFPE* designs and, for that, we designed benchmarks specifically aimed to show the behavior of elasticity in extreme situations. To do so, within the setup stage, we pre-inserted all 8^8 keys in set I and then we measured

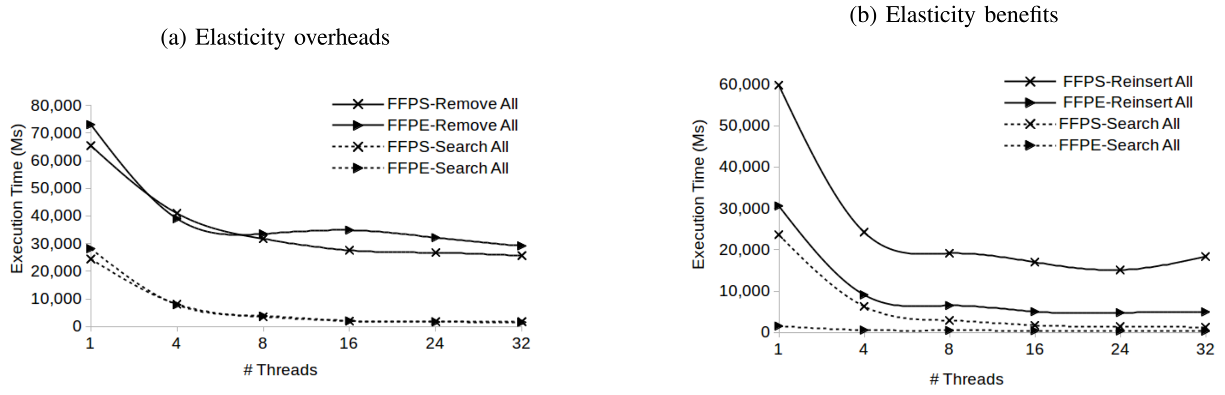

Fig. 6: Elasticity overheads and benefits

the execution time that both designs take to: (i) search for all keys; and (ii) remove all keys.[1] Figure 6a shows the execution time, in milliseconds, for both benchmarks and designs.

The *Search All* benchmark (dashed lines) shows that elasticity has a *negligible or no cost* when the remove operation is not being used. Remember that only the remove operation triggers the hash compression process. On the other hand, the *Remove All* benchmark (solid lines) shows slight differences caused by the hash compression process. With one thread, there is an overhead of 12% (on average, *FFPS* executes in 65,431 ms and *FFPE* in 73,048 ms), but with 4 threads both designs have almost the same execution time. And, as we increase the number of threads, the difference between both designs remains quite stable, ending with a 14% overhead for 32 threads (on average, *FFPS* executes in 25,649 ms and *FFPE* in 29,167 ms).

B. *Elasticity Benefits*

Next, we study the benefits of elasticity in extreme situations. Again, within the setup stage, we pre-inserted all 8^8 keys in set I but then we also remove them, i.e., the *FFPS* has all keys removed but keeps its hash hierarchy unchanged, while the *FFPE* design has all keys and hashes removed (except the root hash). We then measured the execution time that both designs take to: (i) search for all keys; and (ii) reinsert all keys. Figure 6b shows the execution time, in milliseconds, for both benchmarks and designs.

The *Search All* benchmark (dashed lines) shows the potential benefits of the *FFPE* design. With one thread, *FFPE* is *about 15 times faster* than *FFPS* (on average, *FFPS* executes in 23,607 ms and *FFPE* in 1,548 ms). This difference reflects the fact that *FFPS* has to traverse paths of hash levels with depth 8 to verify that a key is missing, while *FFPE* only needs to consult the root hash level. As we increase the number of threads, the memory caches becomes more efficient and *FFPS* is able to reduce its difference. However, with 32 threads, *FFPE* is still *about 4 times faster* than *FFPS* (326 ms and 1,253 ms, respectively).

[1]Since we are using 8 bucket entries per hash level, all chain nodes will be located in a hash level with depth 8.

For the *Reinsert All* benchmark (solid lines), one can observe that the results seem to be consistent with the results from the previous benchmark. The execution times are higher because this benchmark requires reinserting all keys. With one thread, *FFPE* is *about 2 times faster* than *FFPS* (on average, *FFPS* executes in 59,758 ms and *FFPE* in 30,582 ms) and, as we increase the number of threads, the difference consistently increases, such that with 32 threads, *FFPE* is *about 4 times faster* than *FFPS* (4,965 ms and 18,355 ms, respectively). This shows that elasticity is a good strategy even if we have to reinsert hashes when reinserting keys.

C. *Comparison Against Other Designs*

This subsection presents experimental results comparing our design against other state-of-the-art concurrent hash map designs. Here, our initial setup creates an even distribution of keys by the different hash level depths, such that, each remove, search and insert operation has an equal probability of $\frac{1}{8}$ of traversing a path with depth d ($1 \leq d \leq 8$). To do so, we begin by inserting all 8^8 keys in set I and then we remove $8^8 - 8^7$ of those keys, leaving the hash map with 8^7 (2,097,152) keys evenly distributed by the 8 hash level depths. We then measured the execution time that all designs take to run different benchmark sets with different pre-defined ratios of remove, search and insert operations.

Table II presents the execution time results and speedups obtained when running the *CSL*, *CT*, *FFPS* and *FFPE* designs on six benchmark sets with different ratios of remove, search and insert operations for 1, 4, 8, 16, 24 and 32 threads.

The 1st and 2nd benchmarks perform only search and insert operations, respectively. The 3rd benchmark splits the remove and search operations in half, and the 4th benchmark, splits evenly the ratios of the operations. The remaining benchmark sets aim to provide a more detailed perspective of the behavior of the designs as we decrease the weight of remove operations.

For the 1st benchmark (remove: 0% search: 100% insert: 0%), the *CSL* design shows the worst results with an execution time of 54,850 ms and, as we increase the number of threads, it keeps a higher execution time when compared to the other designs. As expected, *FFPE* is by far better than *FFPS*, and

TABLE II: Execution time, in milliseconds, and speedup ratio (against one thread) for running the *CSL*, *CT*, *FFPS* and *FFPE* designs with 1, 4, 8, 16, 24 and 32 threads, on 6 benchmarks with different ratios of remove, search and insert operations

#	Execution Time (E_{T_p})				Speedup Ratio (E_{T_1}/E_{T_p})			
	CSL	**CT**	**FFPS**	**FFPE**	**CSL**	**CT**	**FFPS**	**FFPE**
1st – remove: 0% search: 100% insert: 0%								
1	54,850	14,720	25,529	9,511				
4	15,221	4,293	6,650	2,154	3.60	3.43	3.84	4.42
8	7,825	2,093	3,021	1,282	7.01	7.03	8.45	7.42
16	4,807	1,251	1,804	859	11.41	11.77	14.15	11.07
24	4,773	990	1,448	733	11.49	14.87	17.63	12.98
32	4,428	904	1,570	631	12.39	16.28	16.26	15.07
3rd – remove: 50% search: 50% insert: 0%								
1	52,188	16,008	25,874	9,801				
4	15,656	4,699	6,552	2,444	3.33	3.41	3.95	4.01
8	8,544	2,399	3,263	1,480	6.11	6.67	7.93	6.62
16	5,591	1,524	2,023	1,108	9.33	10.50	12.79	8.85
24	5,274	1,280	1,415	945	9.90	12.51	18.29	10.37
32	5,188	1,344	1,768	952	10.06	11.91	14.63	10.30
5th – remove: 40% search: 40% insert: 20%								
1	76,120	21,843	30,589	21,690				
4	23,187	6,414	7,700	5,685	3.28	3.41	3.97	3.82
8	12,511	3,515	3,980	3,156	6.08	6.21	7.69	6.87
16	7,875	2,386	2,629	1,998	9.67	9.15	11.64	10.86
24	7,906	2,209	2,452	1,779	9.63	9.89	12.48	12.19
32	7,027	2,200	2,333	1,791	10.83	9.93	13.11	12.11
2nd – remove: 0% search: 0% insert: 100%								
1	100,033	36,781	48,321	31,666				
4	30,646	11,740	16,992	9,265	3.26	3.13	2.84	3.42
8	16,089	7,119	11,048	5,537	6.22	5.17	4.37	5.72
16	9,903	5,341	9,983	3,871	10.10	6.89	4.84	8.18
24	9,191	4,980	9,083	3,691	10.88	7.39	5.32	8.58
32	8,636	4,838	9,177	3,923	11.58	7.60	5.27	8.07
4th – remove: 33% search: 33% insert: 33%								
1	77,543	23,910	35,272	24,115				
4	25,418	7,116	8,354	6,681	3.05	3.36	4.22	3.61
8	13,811	4,163	4,785	3,776	5.61	5.74	7.37	6.39
16	9,093	3,038	3,131	2,518	8.53	7.87	11.27	9.58
24	7,974	2,681	2,918	2,484	9.72	8.92	12.09	9.71
32	8,444	2,552	3,038	2,428	9.18	9.37	11.61	9.93
6th – remove: 20% search: 40% insert: 40%								
1	82,145	25,061	34,771	26,087				
4	25,789	7,859	8,620	6,972	3.19	3.19	4.03	3.74
8	13,898	4,373	4,865	3,915	5.91	5.73	7.15	6.66
16	8,659	3,047	3,441	3,043	9.49	8.22	10.10	8.57
24	8,514	2,877	3,144	2,694	9.65	8.71	11.06	9.68
32	6,854	2,773	3,096	2,385	11.98	9.04	11.23	10.94

it is also the best, performing also better than *CT*. *FFPE* maintains a steady difference to *CT* even when we increase the number of threads.

For the 2nd benchmark (remove: 0% search: 0% insert: 100%), again, *CSL* is the design with the worst results, having an execution time of 100,033 ms. However, as we increase the number of threads, *CSL* is able to revert the difference to *FFPS* (the second worst) reaching, with 32 threads, an execution time of 8,636 ms against 9,177 ms, respectively. On the other hand, *FFPE* is still the design with the lowest execution time in all thread launches, it executes in 31,666 ms with one thread, and keeps decreasing, as we increase the number of threads, until it reaches the best execution time with 24 threads, performing 3,691 ms.

For the 3rd benchmark (remove: 50% search: 50% insert: 0%), *CSL* is still the design with the worst performance, having an execution time of 52,188 ms and it is not able to reduce the difference to the other designs as we increase the number of threads. Additionally, *FFPE* has again the best performance in all thread launches, having an execution time of 9,801 ms and 952 ms for 1 and 32 threads, respectively. For the remaining benchmarks (4th to 6th), it is interesting to notice that *FFPE* costs with the remove operation are pretty much compensated by the benefits with the search operation, the backbone procedure of the insert and remove operations.

Finally, for the speedups, one can observe a similar tendency in all designs, with some advantage to *CSL* when the ratio of inserts is higher and to *FFPE* and *CT* when the ratio of searches is higher. In any case, since, in general, *FFPE* starts from lower execution times with one thread, the other designs have more space to achieve better speedups.

VI. Conclusions and Further Work

We have presented a novel, scalable and elastic hash trie design that fully supports the concurrent search, insert, remove, expand and compress operations. To the best of our knowledge, this is the first concurrent hash map design that puts together being lock-free, using fixed size data structures with persistent memory references, sorted keys and be elastic.

Experimental results show that elasticity overheads are largely overcome by its benefits and that it effectively improves the search operation, and, by doing so, our design became very competitive, when compared against other state-of-the-art designs implemented in Java.

As further work, we plan to use our design as the building block for a novel *distributed hash map* design.

References

[1] P. Bagwell, "Ideal Hash Trees," *Es Grands Champs*, vol. 1195, 2001.

[2] D. P. Mehta and S. Sahni, *Handbook of Data Structures and Applications*. Chapman & Hall/CRC, 2004.

[3] R. Grossi and G. Ottaviano, "Fast compressed tries through path decompositions," *Journal of Experimental Algorithmics*, vol. 19, 2015.

[4] Y. Sagiv, "Concurrent operations on B*-trees with overtaking," *Journal of Computer and System Sciences*, vol. 33, no. 2, pp. 275 – 296, 1986.

[5] T. Brown, "Techniques for Constructing Efficient Lock-free Data Structures," Ph.D. dissertation, University of Toronto, 2017.

[6] M. Areias and R. Rocha, "On Extending a Fixed Size, Persistent and Lock-Free Hash Map Design to Store Sorted Keys," in *International Symposium on Parallel and Distributed Processing with Applications*, M. Dong, R. Ranjan, M. Cafaro, and W. Wang, Eds. Melbourne, Australia: IEEE Computer Society, December 2018, pp. 415–422.

[7] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *International Conference on Distributed Computing*, ser. DISC '01. Springer-Verlag, 2001, pp. 300–314.

[8] M. M. Michael, "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets," in *ACM Symposium on Parallel Algorithms and Architectures*. ACM, 2002, pp. 73–82.

[9] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A Provably Correct Scalable Concurrent Skip List," in *International Conference on Principles of Distributed Systems, Technical Report*, Bordeaux, France, 2006.

[10] O. Shalev and N. Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379–405, 2006.

[11] E. Fredkin, "Trie Memory," *Communications of the ACM*, vol. 3, pp. 490–499, 1962.

[12] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent Tries with Efficient Non-Blocking Snapshots," in *ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 151–160.