Computing manuscript No.
(will be inserted by the editor)

On the Correctness of a Lock-Free Compression-based Elastic Mechanism for a Hash Trie Design

Miguel Areias · Ricardo Rocha

Received: date / Accepted: date

Abstract A key aspect of any hash map design is the problem of dynamically resizing it in order to deal with hash collisions. Compression in tree-based hash maps is the ability of reducing the depth of the internal hash levels that support the hash map. In this context, *elasticity* refers to the ability of automatically resizing 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 map 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 is adjusted, as closely as possible, to the set of keys that is stored in the data structure. To materialize our design, we introduce a new compress operation for hash levels, which requires redesigning the existing search, insert, remove and expand operations in order to maintain the lock-freedom property of the data structure. 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.

Keywords Data Structures · Lock-Freedom · Hash Tries

This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. Miguel Areias was funded by the FCT grant SFRH/BPD/108018/2015.

Miguel Areias
Faculty of Sciences – University of Porto
E-mail: miguel-areias@dcc.fc.up.pt

Ricardo Rocha
Faculty of Sciences – University of Porto
E-mail: ricroc@dcc.fc.up.pt

1 Introduction

Organizing data according to a specific layout or schema, making it more presentable and/or accessible, can be a complex task. Such a task often involves identifying uniquely a portion of the data through the use of a key (or an index) and then using the key to quickly map and access the specific data without having to search the entire dataset [**?**]. This procedure is widely implemented by organizing data into key/content pairs and by using optimized techniques to store keys in data structures where they can be quickly searched for. When a key is found, the content's field indicates where the data is stored.

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 [**?**]. 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.

The advantages of compressing tree-based data structures are well-known in the literature [**?**,**?**]. Compression can be performed gradually or incrementally at shallow or deeper tree levels, thus affecting just a small part of the entire data structure, but a key advantage is that it can be done concurrently with the other operations. Two good examples are: (i) the B*-tree proposal [**?**], 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 [**?**] 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 of automatically resizing 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. Operations of tree-based hash-maps take $\mathcal{O}(\log_E K)$ time to complete, 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 hash 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 map design, named *Free Fixed Persistent Hash Map with Elasticity (FFPE)*, that puts together the following characteristics: (i) lock-freedom; (ii) fixed size data structures; (iii) persistent memory references; (iv) sorted keys; and (v) elasticity. Our design is based on hash tries to implement fixed size data structures with persistent

memory references, on single-word CAS (compare-and-swap) instructions to implement lock-freedom, and on *xor* operations to assist in sorting the hash values corresponding to keys.

In previous work [**?**], Areias and Rocha proposed a concurrent hash map design, named *FFPS*, 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. Recently, Moreno et al. [**?**] presented an alternative lock-free compression design for a similar lock-free trie-based hash map that was able to significantly reduce the depth of the internal hash levels within the hash map structure, by *swapping* multiple shallow hash levels for a single hash level that is extended enough (i.e., has a sufficient amount of bucket entries) to hold all of the buckets entries of the swapped shallow hash levels. By doing so, the lock-free compression design was able to minimize cache misses and increase the overall throughput of the default search, insert and remove operations.

Table 1: Characteristics support

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

To the best of our knowledge, none of the available designs in the literature fulfills all the above five characteristics simultaneously. Table 1 shows how the *FFPE* and *FFPS* designs compare with two other state-of-the-art designs, the *Concurrent Skip-Lists (CSL)* from the Java concurrency package [**?**] and the *Concurrent Tries (CT)* [**?**,**?**], regarding those characteristics.

To materialize our design, we introduce a new *compress operation* for hash levels, which required adapting freezing (and unfreezing) based procedures [**?**,**?**,**?**], and required redesigning the existing search, insert, remove and expand operations in order to maintain the lock-freedom property of the whole design. Experimental results show that elasticity reduces the number of hash levels and, in doing so, it effectively improves the performance of the search operation, which is the backbone procedure for all other operations. Consequently, *FFPE* became very competitive when compared to the *CSL* and *CT* designs. In a nutshell, the main contributions of this work are:

- We present and discuss the key algorithms required to easily reproduce our implementation by others;
- We present a proof of correctness showing that our proposal is linearizable and lock-free for the search, insert and remove operations;
- We present a set of experiments comparing the benefits and drawbacks of elasticity, and a comparison analysis of our design against other state-of-the-art concurrent hash map designs.

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 and we discuss their correctness. 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.

2 Background

Nowadays, the CAS instruction is at the heart of many lock-free data structures [**?**]. A lock-free data structure guarantees that, whenever a thread executes some finite number of steps, at least one operation on the data structure by some thread must have made *progress* during the execution of these steps. Herlihy and Shavit proposed a *grand unified explanation* [**?**] for the progress properties using *linearizability*, which is an important correctness condition for the implementation of concurrent data structures [**?**]. The first correct CAS-based lock-free list-based set design was introduced by Harris [**?**]. 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 [**?**]. Skip lists are 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.* [**?**] and, Shalev and Shavit [**?**].

Sorted search trees are important in systems that require indexing by keys, such as, database systems. Several examples exist that show the importance of these trees, for instance, B-Trees were proposed to organize large ordered sets [**?**,**?**], and T-Trees have been proposed as a better index structure in main memory database systems [**?**]. Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash tables [**?**]. An essential property of the trie data structure is that common prefixes are stored only once [**?**], which in the context of hash tables allows us to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions. Prokopec *et al.* presented the CTries [**?**,**?**], 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. And more recently, Brown et al. presented the C-IST [**?**], a specialized trie-based data structure that is the first non-blocking implementation of the classic interpolation search tree [**?**].

Traditional hash maps do not store sorted keys, which makes them unsuitable for non-exact match queries, such as, to find all keys in an interval. However, they are known for their excellent performance in searching for items.

Searching is a crucial time-consuming part of many applications, and using a good search method instead of a bad one often leads to a substantial increase in performance [**?**]. The earliest search algorithm – binary search – was first mentioned by John Mauchly [**?**] more than six decades ago, 25 years before the advent of relational databases [**?**]. Later, Peterson proposed interpolation search [**?**], an optimized version of binary search, that, instead of choosing the middle element for splitting the search, chooses the splitting element according to the value of the key being searched. Nowadays, one of the most critical database primitives is tree-structured index search, which is used for a wide range of applications where low latency and high throughput matter, such as, data mining, financial analysis, scientific workloads, among others [**?**].

More recently, Areias and Rocha presented a novel trie-based lock-free hash-map design, named *FFPS*, that combines hashing with sort and tree search algorithms to support additional important properties, such as, fixed-size data structures, persistent references and sorted keys [**?**]. Fixed size data structures have pre-defined sizes, which allows to efficiently solve the problems of setting the size of the initial hash map and of dynamically expanding/resizing it in order to deal with hash collisions, and it is also very important to take advantage of memory allocators where data structures of the same size/class are (pre-)allocated within the same (regions of) pages [**?**]. Persistent references pin memory references to information stored in a specific data structure. By doing so, it avoids duplicating internal data structures by creating new ones through copying/removing the older ones. The persistent characteristic is very important in hash maps that are used not standalone but as a component of a bigger module/library which, for performance reasons, requires accessing directly the internal data structures.

At the implementation level, the *FFPS* 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} chunk of w bits of h to index the entry in the appropriate bucket array of H_i. The most significant chunk of w bits represents the first level and the least significant chunk of w bits is the last level. 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, the *FFPS* design 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.

3 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. We first intro-

duce how the insertion of nodes triggers the expansion of hash levels and then we present how the removal of nodes triggers the compression of hash levels.

3.1 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 consists in a bucket array of 2^w entries and 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*.

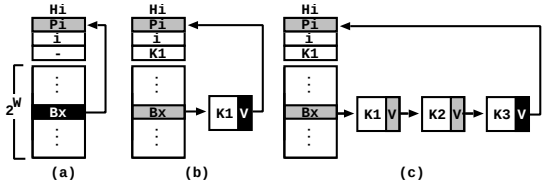


Fig. 1: Insert operation in a hash level

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.

To better understand the figures, the different elements in a hash level are colored accordingly to their condition. A black element, which we name an *Interest Point (IP)*, represents a memory address that can be updated concurrently, during the execution of the hash-map. To guarantee lock-freedom, all updates to black elements are done using CAS operations. A gray element can be updated only by a single thread, but can become an IP at any instant of execution. A white element will no longer be updated.

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, we apply a *xor* operation between the hash values of the key being inserted and the key representative of the hash level, to check in which level (chunk of bits) they differ (remember that the most significant chunk of w bits represents the

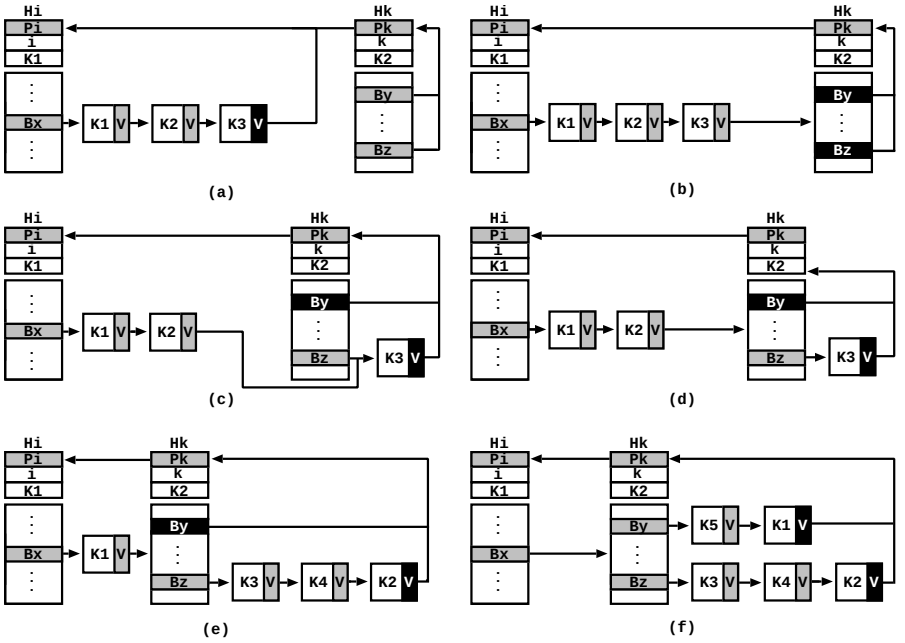


Fig. 2: Front-expansion with the concurrent insertion of nodes

first level). If they differ in a higher level than the hash level identifier, then we insert a new hash level in a deeper level (we call this *front-expansion*). Otherwise, we insert a new hash level in a shallow level (we call this *back-expansion*) [**?**].

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 this case, T starts by pre-allocating a second level hash H_k, with all entries referring the respective level and with a key representative (K_2 in Fig. 2) consisting of the key in the chain that differs in the lowest level from the key being inserted by T.

The new hash level H_k is then used to implement a synchronization point with the current IP (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 the IP and, in this 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.[1] 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 entries. In order to ensure lock-free synchronization, we need to guarantee that, at any time, all threads are 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 passes to the gray state and K_3 becomes the next IP for the insertion of new nodes on B_z. Note 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.

We next describe the *back-expansion* operation, where a new hash levels is inserted in a shallow level. Figure 3 shows a possible sequence of steps involving a back-expansion operation concurrently with a front-expansion operation. The back-expansion operation has a lower priority than the front-expansion operation, i.e., a back-expansion only begins if no front-expansion is undergoing. If a front-expansion is undergoing and a thread T wants to execute a back-expansion, T begins by assisting the threads doing the front-expansion and only then it begins the back-expansion.

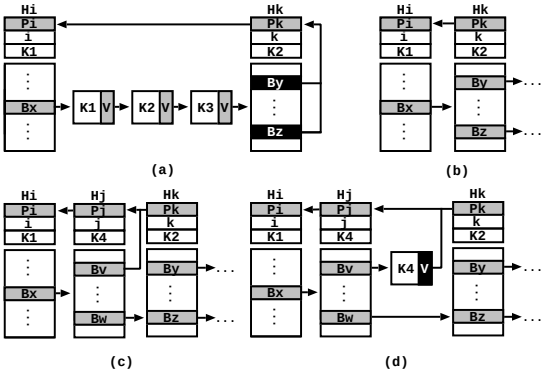


Fig. 3: Concurrent back-expansion

[1] Note that with persistent memory, keys must remain pinned to their initial memory references, i.e., the copy of a key to a new memory reference is not allowed.

Figure 3(a) shows the initial configuration, where a front-expansion operation is undergoing for the bucket entry B_x to a second level hash H_k, and Fig. 3(b) shows the end of the front-expansion, after the CAS operation updating B_x to H_k. Assume now that during the front-expansion, a new thread T reaches B_x looking to insert a key K_4 and that K_4 differs from K_2 in a level j that is lower than k. This means that a third hash level H_j must be included between H_i and H_k. Figure 3(c) shows the resulting configuration after the insertion of the hash level H_j. The initialization of a hash level in a back-expansion is sightly different from the front-expansion, since one of the bucket entries must refer the front hash level H_k. T uses the key representative of the front hash level, K_2 in Fig. 3(c), to compute the bucket entry B_w that should refer to H_k. All the remaining bucket entries are initialized referring back the respective level H_j. At the end of the initialization step, T applies a CAS operation on B_x setting it to refer to H_j. Finally, T can insert the key K_4 in the hash level H_j. Figure 3(d) shows this final configuration.

3.2 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. 4 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. 4(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. 4(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 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.

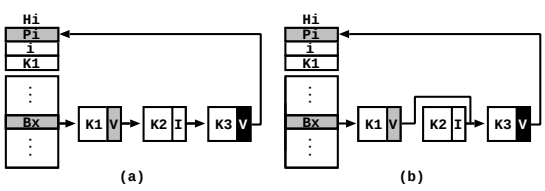


Fig. 4: Remove operation in a hash level

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 5 illustrates an example of a successful compress operation.

Figure 5(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. 5(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. 5(c)).

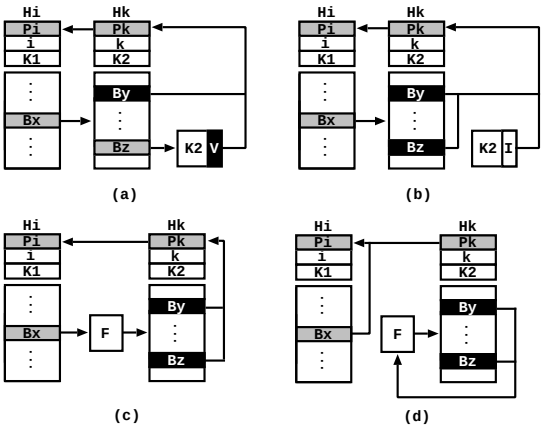


Fig. 5: A successful compress operation

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. 5(d)).

It is important to notice 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. 5 and assume that a second thread T_2 is preempted in H_k, at the time of the configuration in Fig. 5(c). Later, if T_2 is resumed after the configuration in Fig. 5(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. 5(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 this 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 6 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.

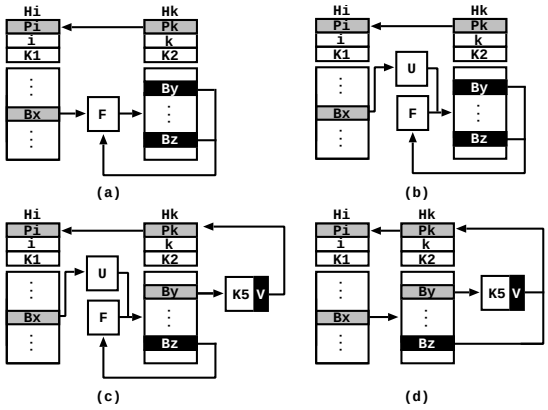


Fig. 6: Aborting a compress operation

Figure 6(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. 6(b)). Once this notification succeeds, T_2 proceeds by inserting K_5 in B_y (Fig. 6(c)). It is important to notice that if another thread T_3 is 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 6(d) shows the corresponding final configuration.

4 Algorithms

This section presents the most relevant algorithms of our design.

4.1 The *SearchRemoveKey* Operation

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

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

```
 1: B ← GetBucket(K, H)
 2: R ← NextRef(B)
 3: if IsCompressionNode(R)
 4:   R ← NextRef(R)
 5: if IsHashLevel(R)
 6:   if R = H    // 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) ∧ Key(R) = K    // key found
12:     if MakeChainNodeInvalid(R)
13:       return MakeChainNodeUnreachable(R, H)
14:   R ← NextRef(R)
15: until IsHashLevel(R)
16: if R = H    // chain ended in the same hash level
17:   return
18: R ← GetHashLevel(R, Level(H) + 1)
19: return SearchRemoveKey(K, R)
```

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 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).

4.2 The *MakeChainNodeUnreachable* Operation

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.

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

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

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). During the process, it saves in BRN the chain reference of BR (line 10).

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 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 this 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).

4.3 The *FreezeHashLevel* & *UnfreezeHashLevel* Operations

Algorithm 3 *FreezeHashLevel*(*Node* F, *Hash* H)

```
1: for B in BucketEntries(H)
2:   if not CAS(NextRef(B), H, F)
3:     return false
4: return true
```

The allocation and insertion of a freezing compression node marks the beginning of the freezing procedure. Algorithm 3 shows the process of trying to freeze a hash level H, i.e., mark all bucket entries of H as ready for compression, given a freezing compression node F. For each bucket entry B of H, the algorithm applies a CAS operation trying to update the entry's reference to node F. If one of the bucket entries is not empty, the CAS fails and the algorithm immediately returns *false*, signaling that the procedure has failed. Only when all CAS operations succeed, i.e., all bucket entries are made to refer to node F, the algorithm returns *true*.

As an optimization, we kept track of the (non empty) bucket entry B where the procedure has failed in such a way that the next freezing procedure for the current hash level H only takes place if it is B to trigger the process (in this way we avoid starting the freezing procedure knowing that there is, at least, one non-empty bucket entry B). For simplicity of presentation, we are not including this optimization in the algorithms being presented.

Algorithm 4 then shows the process of unfreezing a hash level H, which is triggered when the thread doing a compress operation detects the existence of an unfreezing compression node. This requires traversing again all bucket entries of H in order to update the ones referring a given freezing compression node F, i.e., for each bucket entry B of H, the algorithm applies a CAS operation trying to update the entry's reference to H if it is referring F, thus restoring the previous configuration.

Algorithm 4 *UnfreezeHashLevel*(*Node* F, *Hash* H)

```
1: for B in BucketEntries(H)
2:   CAS(NextRef(B), F, H)
3: return
```

4.4 The *CompressHashLevel* Operation

Finally, Alg. 5 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 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). 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 *UnfreezeHashLevel*() 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

Algorithm 5 *CompressHashLevel*(*Key K*, *Hash H*)

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

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).

5 Correctness & Complexity

In this section, we discuss the correctness and complexity of our design. The full proof consists of two parts: (i) prove that the design is *linearizable*; and (ii) prove that the *lock-freedom* property holds in all operations. Due to the lack of space, in what follows, we focus on the linearization proof for the algorithms described before. For that, we enumerate the linearization points, describe the set of invariants and show parts of the proof that the linearization points preserve the set of invariants.

5.1 Linearization Points

The linearization points in the algorithms shown are:

LP_1 *SearchRemoveKey()* (Alg. 1) is linearizable at successful procedure *MakeChainNodeInvalid()* in line 12.
LP_2 *MakeChainNodeUnreachable()* (Alg. 2) is linearizable at successful CAS in line 15.
LP_3 *MakeChainNodeUnreachable()* (Alg. 2) is linearizable at successful CAS in line 20.
LP_4 *FreezeHashLevel()* (Alg. 3) is linearizable at successful CAS in line 2.
LP_5 *UnfreezeHashLevel()* (Alg. 4) is linearizable at successful CAS in line 2.
LP_6 *CompressHashLevel()* (Alg. 5) is linearizable at successful CAS in line 6.
LP_7 *CompressHashLevel()* (Alg. 5) is linearizable at successful CAS in line 14.
LP_8 *CompressHashLevel()* (Alg. 5) is linearizable at successful CAS in line 19.
LP_9 *CompressHashLevel()* (Alg. 5) is linearizable at successful CAS in line 27.

5.2 Invariants

The set of invariants that must be *preserved* on every state of the data structure are:

Inv_1 Given a hash level H, *PrevHashLevel*(H) always refers to a hash level PH which is previous to H.
Inv_2 A hash level H must comply with the following semantics: (i) its initial state is valid; (ii) after a successful compression, its state turns invalid and never changes to valid again. While valid, all bucket entries are changeable, whereas in a invalid state, all bucket entries are unaltered.
Inv_3 A bucket entry B belonging to a hash level H must comply with the following semantics: (i) its initial reference is H; (ii) after the first update, it must refer to a chain node N_1 or to a compression node C_1; (iii) after a follower update, it must refer to a different chain node N_2, to a different compression node C_2, to the hash level H or to a deeper hash level H_d such that *PrevHashLevel*(H_d) = H.
Inv_4 A compression node C, of type *Freeze* or *Unfreeze*, must comply with the following semantics: (i) its initial reference must refer to a hash level H under compression and inserted in a bucket entry B within the previous hash level PH; (ii) the reference in C remains unaltered; (iii) if C is reachable then H is valid; (iv) if C is of type *Unfreeze* then H is valid; (v) if C is of type $Freeze$ and unreachable then H is invalid; (vi) if H is invalid then C is reachable from all bucket entries in H.
Inv_5 A chain node N must comply with the following semantics: (i) its initial state is valid; (ii) after an invalidation step, its state turns invalid and never changes to valid again.
Inv_6 Given an invalid chain node N in a chain of nodes starting from a bucket entry B belonging to a hash level H, N must comply with the following semantics: (i) at any given instant, it exists only one point I (the bucket entry B or a valid chain node) from where N should be made unreachable; (ii) if N is unreachable then its reference is never updated again.

5.3 Proofs

Next, we show the proof on how two of the linearization points, namely $\boldsymbol{LP_5}$ and $\boldsymbol{LP_7}$, preserve the set of invariants. The remaining linearization points follow a similar proof strategy.

Lemma 1 *LP_5 preserves the set of invariants.*

Proof After the execution of the CAS operation in line 2, the reference in the bucket entry B either changes from the compression node F to the hash level H (if the CAS operation succeeds) or remains unaltered (if the CAS operation fails). In both situations, invariant Inv_3 holds. The remaining invariants are not affected. □

Lemma 2 *LP_7 preserves the set of invariants.*

Proof Consider that H is the hash level that is under compression, PH is the previous hash level, B is a bucket entry within PH and F is a compression node with type *Freeze* and referring H. This linearization point concerns the final stage of the compress operation, thus it affects invariants Inv_2, Inv_3 and Inv_4. The remaining invariants are not affected.

Previous to the execution of the CAS in line 14, invariant Inv_2 ensures that H is valid. Then, if the CAS fails, H remains valid. Otherwise, if the CAS succeeds, H is no longer reachable form PH. For the CAS to be executed, the *FreezeHashLevel*() operation in line 7 must have succeeded, thus H is for sure in an invalid state and invariant Inv_2 holds.

Regarding the reference in B, it either changes to PH (if the CAS succeeds) or remains unaltered (if the CAS fails). In both situations, invariant Inv_3 holds.

If the CAS operation fails, that means that F is already unreachable from B. Otherwise, the CAS operation makes F unreachable from B. In both situations, F is reachable from H as a result of the *FreezeHashLevel*() in line 7, and thus invariant Inv_4 also holds. □

5.4 Complexity

Our design combines the identity hash function with a hierarchy of hash levels whose branching factor is given by a fixed (and pre-defined) number of bucket entries per hash level and whose maximum depth (or height) depends on the overall number of keys inserted in the hash map. As show in the previous sections, our design supports multiple keys per bucket entry in the hash trie map leaves, e.g., in Fig. 1, we showed three keys associated to a single bucket. However, for the sake of simplicity, next we will formalize the complexity of our design, assuming an evenly distribution of keys, that generate the worst configuration possible for memory usage, which is to associate each key to a single bucket, i.e., all chains of nodes holding the keys, will have only one node. This is formalized next.

Lemma 3 *Given a fixed number E of bucket entries per hash level and an overall number K of keys inserted in the hash map, the average space complexity on the number of hashes used is $\mathcal{O}(\frac{K}{E})$.*

Proof Assuming that K keys generate a perfect hash trie map, i.e., all internal hash levels have E children, all leaves are at the same level and all bucket entries in the leaves are referring to different keys, then the total number of hashes is given by $T = \sum_{h=0}^{H} E^h = \frac{E^{H+1}-1}{E-1}$, where H is the height of the hash trie map. On the other hand, the number of leaves is given by $L = E^H$, and since each key is associated to a bucket entry in the leaves, then $K = E * L$, which means that T can be rewritten as $T = \frac{K-1}{E-1} \approx \frac{K}{E}$. □

Lemma 4 *Given a fixed number E of bucket entries per hash level and an overall number K of keys inserted in the hash map, the time complexity (average depth) on the number of hashes traversed is $\mathcal{O}(\log_E K)$.*

Proof Assuming that K keys generate a perfect hash trie map, i.e., all internal hash levels have E children, all leaves are at same level and all bucket entries in the leaves are referring to different keys, then the total number of leaves L is given by $L = E^H \Leftrightarrow H = \log_E L$, where H is the height of the hash trie map. On the other hand, since each key is associated to a bucket entry in the leaves, $K = E * L \Leftrightarrow L = \frac{K}{E}$, which means that H can be rewritten as $H = \log_E \frac{K}{E} = \log_E K - \log_E E = \log_E K - 1 \approx \log_E K$. □

6 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.

For the experiments, we developed an open source benchmarking environment that contains different 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 operations, we equally divide the size of S by the number T of running threads and allow each thread to run $\frac{S}{T}$ randomized operations, such that all threads execute a similar ratio of operations. To support scalable and non concurrent random number generation on each thread, we used JVM's *ThreadLocalRandom*. Additionally, we configured the benchmarking environment 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) 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.

In the next two subsections, we focus in understanding the overheads and the benefits of the elasticity mechanism. To do so, we will be comparing the *FFPS* and *FFPE* designs in worst-case and best-case scenarios. We ran both designs with 8 buckets entries per hash level, a threshold value of 2 chain nodes for the hash collisions, and implementing sorted keys. Finally, in the last subsection, we focus in comparing our design against other state-of-the-art concurrent hash map designs.[2]

6.1 Elasticity Overheads

We begin with a set of benchmarks specifically designed 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 the execution time that both designs take to: (i) search for all keys; and (ii) remove all keys.[3] Figure 7a 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. 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*). Given the number of remove operations in this benchmark, we argue that these are very acceptable results. Note that the compression process is also running in situations where the hash levels may not be empty and in situations where more than one thread is trying to compress the same hash level.

Additionally to the execution time, we measured the number of hashes successfully compressed with different number of threads launches (including the base execution with one thread) and, in general, the number of hashes compressions was quite similar, i.e., in almost all thread launches, the number of empty hashes at the end of execution (not compressed hashes) was zero or closer to zero, as expected by Lemma 3 and Lemma 4.

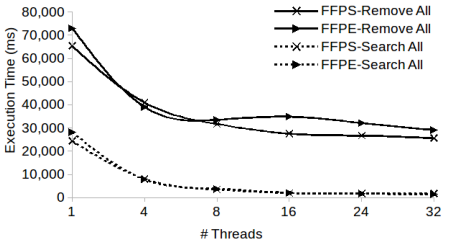
[2] The designs and the benchmarks are available at https://github.com/miar/ffpe

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

6.2 Elasticity Benefits

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 7b shows the execution time, in milliseconds, for both benchmarks and designs.

(a) Elasticity overheads in worst-case scenario



(b) Elasticity benefits in best-case scenario

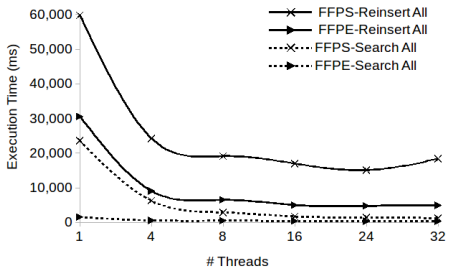


Fig. 7: Elasticity overheads and benefits

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).

Memory cache effects are really important to understand these results. Remember that we are generating random keys in set I. We have also experimented with sequential keys, i.e., search for keys sequentially instead of randomly, and, in that case, the gains obtained with elasticity were almost neglectable (sequential keys traverse the same hashes, making it an ideal scenario for memory caches).

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.

In summary, elasticity effectively improves the performance of the search operation, because it adjusts the number of levels in a path to match the current demand as stated by Lemma 3 and Lemma 4. The search operation is the backbone procedure of both insert and remove operations, since to insert or remove something, the search operation must be executed first. Our experimental results show that, by reducing the number of hash levels to be traversed, we are able to significantly improve the execution time of the search operation. And, by doing so, we argue that the overheads of elasticity are insignificant when compared to its benefits.

6.3 Comparison Against Other Designs

This subsection presents experimental results comparing our design against other state-of-the-art hash map designs, namely the *Concurrent Skip-Lists (CSL)* from the Java's concurrency package and the *Concurrent Tries (CT)* as proposed by Prokopec *et al.*.

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 2 presents the execution time results and speedups obtained when running the *CSL*, *CT*, *FFPS* and *FFPE* designs on eight benchmark sets with different ratios of concurrent remove, search and insert operations for 1, 4, 8, 16, 24 and 32 threads.[4] 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 the remove operations. The 5th benchmark has 40% remove operations, while the 6th benchmark has 20%. The 7th and 8th benchmarks only have 10% remove operations but differ on the search and insert ratios.[5]

[4] We are not including memory usage results since we were not able to obtain meaningful results from JVM about the memory footprints of the several designs. We used the formula *'Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()'* but the results obtained were not accurate, with no good reasons to have big differences across the different runs of the same design.

[5] The approximated number of keys in the data structures at the end of each benchmark is approximated by the expression $8^8 * (RS + RI)$, where RS and RI are the ratios of the keys searched and the keys inserted, respectively, described in each benchmark.

Table 2: Execution time (average of 10 runs), 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 8 benchmarks with different ratios of concurrent 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
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
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
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
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
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
7th − remove: 10%			**search: 70%**		**insert: 20%**			
1	72,679	**21,071**	29,106	21,418				
4	22,452	6,305	7,564	**5,598**	3.24	3.34	**3.85**	3.83
8	11,990	3,358	4,039	**2,993**	6.06	6.27	**7.21**	7.16
16	7,872	2,217	2,560	**1,954**	9.23	9.50	**11.37**	10.96
24	7,265	2,097	2,297	**1,617**	10.00	10.05	12.67	**13.25**
32	7,426	1,913	2,234	**1,581**	9.79	11.01	13.03	**13.55**
8th − remove: 10%			**search: 10%**		**insert: 80%**			
1	99,995	35,118	41,598	**31,841**				
4	30,840	10,144	12,984	**8,525**	3.24	3.46	3.20	**3.74**
8	16,332	6,255	8,853	**5,315**	**6.12**	5.61	4.70	5.99
16	9,698	4,867	6,852	**4,008**	**10.31**	7.22	6.07	7.94
24	8,898	4,255	6,291	**3,820**	**11.24**	8.25	6.61	8.34
32	8,519	3,990	6,191	**3,847**	**11.74**	8.80	6.72	8.28

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 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. Both have a similar behavior, their worst execution time is with one thread (9,511 ms and 14,720 ms, respectively), and their best execution time is achieved with 32 threads (631 ms and 904 ms, respectively).

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. *FFPE* is immediately followed by *CT* that is able to approach the *FFPE* execution time with 32 threads (4,838 ms against 3,923 ms, respectively).

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 4th benchmark (remove: 33% search: 33% insert: 33%), *CSL* continues to show the worst results, having an execution time of 77,543 ms, and it is not able to approach the performance of the remaining designs, as we increase the number of threads. *FFPE* has again the best execution time, reaching its best with 32 threads (execution time of 2,428 ms). *FFPE* has a better execution time than *FFPS*, but it is immediately followed by *CT*. The differences are so tight that, with one thread, are almost the same, *CT* achieves an execution time of 23,910 ms against 24,115 ms. However, *CT* loses the best performance to *FFPE* in all remaining thread launches. The biggest difference between both designs is achieved with 16 threads, where *CT* has an execution time of 3,038 ms against 2,518 ms for the *FFPE* design.

For the remaining benchmarks (5th to 8th), it is interesting to notice that *FFPE* costs with the remove operation are pretty much compensated by the benefits with the search operation. Remember that the search operation is the backbone procedure of both insert and remove operations. *FFPE* always achieves the best execution times and, when comparing against *CSL*, which also supports sorted keys, *FFPE* executes in much less time in all benchmarks, with differences being quite significant, in general. Concerning the overall comparison between the *FFPS* and *FFPE* designs, the values in the Table 2 show that *FFPE* outperforms *FFPS* in all configurations, which clearly demon-

strates the potential of the elasticity support and the advantages of the *FFPE* design in terms of execution time when compared to *FFPS*.

For the speedups, one can observe a similar tendency in all benchmarks. *FFPS* seems to scale better for the benchmarks that have a higher ratio of searches (as a result of the memory cache effects discussed in Subsection 6.2), while *FFPE* seems to scale better for the benchmarks that have a higher ratio of inserts. In any case, since *FFPS* starts from higher execution times with one thread, it has more space to achieve better speedups. In general, one can observe that all designs seem to have scalability problems, once the best speedup of all experiments is only 18.29 (obtained for the *FFPS* design with 24 threads on the 3th benchmark). To better understand these results, one must remember that the environment for our experiments was a 2 sockets - 16 cores SMP/NUMA based architecture. A SMP system is a *share everything* system where multiple processors are working under the supervision of a single operating system and all processors access memory using a common bus or inter-connect path. This means that, as we increase the number of processors in the computation, the bus becomes overloaded which can result in a performance bottleneck. NUMA tries to mitigate the burden of the main bus by adding intermediate levels of memory shared among some of the processors so that several data accesses do not need to travel on the main bus. However, on applications that have irregular data requests, as the benchmarks that we are using in these experiments, the efficiency of the intermediate levels of memory is lower and in some situations can even have a negative impact in the performance.[6] As such, in this benchmark environment, the speedups of all designs show 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.

In summary, our novel *FFPE* design showed to be an excellent alternative to the other state-of-the-art hash map designs. In general, the *FFPE* speedups are in line with the competition, but the execution times are, most times, significantly lower than the remaining designs. Moreover, the *FFPE* design supports multiple features that are not supported by their competitors. Two good examples are the support for sorted keys feature (not supported by *CT*), which implements non-exact match queries, such as, finding all keys in a given interval, and the support for elasticity (not supported by *CSL* and *FFPS*), which optimizes memory resources to the current demand.

7 Conclusions and Further Work

We have presented a novel, scalable and elastic hash map 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

[6] SMP/NUMA bottlenecks are analyzed in detail in Drepper's work [**?**].

puts together being lock-free, using fixed size data structures with persistent memory references, sorted keys and be elastic, which we consider to be characteristics that have the best trade-off between performance, correctness and computational environment independence.

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. *FFPE* was able to achieve better execution times than *CT* and *CSL*, in almost all scenarios, and, for some thread launches, the differences are very significant. This is quite an accomplishment if we consider that *CT* is native in Scala (runs on top of JVM) and *CSL* is native in Java's concurrency package.

As further work, we plan to use real world scenarios widely-used in the community, such as, YCSB A-F workloads, Facebook or Twitter synthetic memcached, and compare our design against other designs, such as, the Elastic Cuckoo Page Tables design proposed by Skarlatos et al. [**?**]. We also plan to extend our design to support snapshots, which will allow the usage of iterators like in CTries, and create an extensive user-interface that allows users to do non-exact match queries using iterators.