On the Correctness and Efficiency of a Novel Lock-Free Hash Trie Map Design

Miguel Areias and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{miguel-areias, ricroc}@dcc.fc.up.pt

Abstract

Hash tries are a trie-based data structure with nearly ideal characteristics for the implementation of hash maps. In this paper, we present a novel, simple and scalable hash trie map design that fully supports the concurrent search, insert and remove operations on hash maps. To the best of our knowledge, our proposal is the first that puts together the following characteristics: (i) be lock-free; (ii) use fixed size data structures; and (iii) maintain the access to all internal data structures as persistent memory references. Our design is modular enough to allow different types of configurations aimed for different performances in memory usage and execution time and can be easily implemented in any type of language, library or within other complex data structures. We discuss in detail the key algorithms required to easily reproduce our implementation by others and we present a proof of correctness showing that our proposal is linearizable and lock-free for the search, insert and remove operations. Experimental results show that our proposal is quite competitive when compared against other state-of-the-art proposals implemented in Java.

Keywords: Concurrent Data Structures, Hash Tries, Lock-Freedom

1. Introduction

Hash maps are a very common and efficient data structure used to store and access data that can be organized as pairs (K, C), where K is a unique key with an associated content C. The mapping between K and C is given by a hash function, and the most usual operations done in hash maps are the search, insertion and removal of pairs. Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash maps [1]. An essential property of the trie data structure is that common prefixes are stored only once [2], which in the context of hash maps leads to implementations using *fixed size data structures*. This 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. This fixed size characteristic is also determinant for taking advantage of memory allocators where data structures of the same size/class are (pre-)allocated within the same (regions of) pages [3].

Multithreading with hash maps is the ability to concurrently execute multiple search, insert and remove operations in such a way that each specific operation runs independently but shares the underlying data structures that support the hash map. The traditional approach to concurrent data structures is to use locking primitives such as spinlocks, mutexes or semaphores to synchronize access to critical sections. A fundamental problem with lock-based data structures is that when one thread attempts to acquire a lock held by another thread, the thread needs to block until the lock is available. Blocking a thread is undesirable for many reasons. A non-blocking alternative is to use *lock-free data structures*, as they are unaffected by thread failures or delays. A major difference is that with lock-free data structures, the failure or suspension of any thread cannot cause the failure or suspension of another thread. Lock-freedom can still lead to starvation of individual threads but it guarantees system-wide throughput, i.e., when a set of the program threads are run simultaneously at least one

thread makes progress. In this context, lock-free data structures offer several advantages over their lock-based counterparts, such as, being immune to deadlocks, lock convoying and priority inversion, and being preemption tolerant, which ensures similar performance regardless of the thread scheduling policy. Lock-free data structures have proved to work well in many different settings [4] and they are available in several different frameworks, such as, Intel's Threading Building Blocks [5], the NOBLE library [6] or the Java concurrency package [7].

Another important characteristic is the ability to maintain the access to all internal data structures as *persistent memory references*, i.e., avoid 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. In such scenario, it is mandatory to avoid changing the external memory references to the internal hash map data structures.

In recent work [8], we have proposed a novel concurrent hash map design, aimed to be as competitive as the existent alternative designs, that puts together the following three characteristics: (i) be lock-free; (ii) use fixed size data structures; and (iii) maintain the access to all internal data structures as persistent memory references. To the best of our knowledge, none of the available alternatives in the literature fulfills all these three characteristics simultaneously. In this work, we discuss in detail the key algorithms required to easily reproduce our implementation by others and we present a proof of correctness showing that our proposal is linearizable and lock-free for the search, insert and remove operations. Our proposal is based on single-word CAS (compare-and-swap) instructions to implement lock-freedom and on hash tries to implement fixed size data structures with persistent memory references.

In previous work [9, 10], we have already proposed different concurrent hash map designs but only for the search and insert operations. In [9], we presented a first lock-free hash map design which had two important constrains: (i) it did not use fixed size data structures for the hash tables; and (ii) the expansion mechanism was done by a single thread. In [10], we presented a second lock-free hash map design that overcomes both constrains, for (i) we used hash tries and for (ii) we allowed multiple and simultaneous expansions of the hash levels. However, both designs did not support the concurrent remove operation. The present works revives such previous work and extends it to also include the remove operation. To do so, we had to redesign the existent search, insert and expand operations and add new and more powerful invariants that could ensure the correctness of the new design. An important contribution of the new hash map design is the ability to support the concurrent expansion of hash levels together with the removal of keys without violating the lock-free property and the consistency of the design. Our proposal is also the basis of more recent works in the field [11, 12].

In a nutshell, the main contributions of this work are the following:

- We present and discuss in detail 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 our design against other state-of-the-art concurrent hash map proposals, namely, *Non Blocking Hash Maps* [13], *Concurrent Tries* [14], and the *Concurrent Hash Maps* and *Concurrent Skip-Lists* from the Java concurrency package [7].

The remainder of the paper is organized as follows. First, we introduce relevant background and related work. Next, we present and discuss in detail the key algorithms required to easily reproduce our implementation by others. Then, we present the formal proof that our proposal is linearizable and lock-free. Finally, we present a set of experiments comparing our design against other state-of-the-art concurrent hash map proposals. At the end, we present conclusions and further work directions.

2. Background & Related Work

Nowadays, the CAS instruction can be widely found on many common shared memory architectures. The CAS instruction atomically compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value. The

Assumptions on the OS scheduler

Dependency vs Progress	Non-Blocking Independent	Non-Blocking Dependent	Blocking Dependent
Every thread makes progress	Wait-Free	Obstruction-Free	Starvation-Free
Some thread make progress	Lock-Free	?	Deadlock-Free

Level of Progress

Maximal vs Minimal

Dependent vs Independent

Blocking vs Non-Blocking

Figure 1: The periodic table of progress conditions

atomicity guarantees that the new value is calculated based on up-to-date information, i.e., if the value had been updated by another thread in the meantime, the write would fail. The CAS result indicates whether it has successfully performed the substitution or not.

Besides reducing the granularity of the synchronization, the CAS instruction is at the heart of many lock-free data structures [15]. A lock-free data structure guarantees that, whenever a thread executes some finite number of steps on the data structure, at least one operation by some thread must have made *progress* during the execution of these steps. Progress can be seen as the steps that threads take to complete methods within a concurrent data structure, i.e., the steps that threads take to execute a method between its *invocation* and its *response*. The execution of a concurrent method is then modeled by a *history* H, a finite sequence of method invocation and response events, a *subhistory* of H is a sub-sequence of the events of H and an *interval* is a sub-history consisting of contiguous events. Progress conditions can be placed in a two-dimensional *periodical table*, where one of the axis defines the *assumptions on the OS scheduler* and the other axis defines the *level of progress*. Figure 1 shows the periodic table of progress conditions, as defined by Herlihy and Shavit [16].

Regarding the assumptions on the OS scheduler, progress conditions can follow a *scheduler blocking assumption*, if a thread can block all remaining threads during an access to critical region, or a *scheduler non-blocking assumption*, otherwise. The progress conditions can also follow a *scheduler independent assumption*, which guarantees progress as long as threads are scheduled and no matter how they are scheduled, or a *scheduler dependent assumption*, meaning that the progress of threads rely on the *OS* scheduler to satisfy certain properties. For example, the deadlock-free (threads cannot delay each other perpetually) and starvation-free (a critical region cannot be denied to a thread perpetually) properties guarantee progress, but they depend on the assumption that the *OS* scheduler will let each thread within a critical region to be able to run for a sufficient amount of time, so that it can leave the critical section. Another example is the obstruction-free property [17] (a thread runs within a critical region in a bounded number of steps), which requires the *OS* scheduler to allow each thread to run in isolation for a sufficient amount of time.

Regarding the level of progress, a method provides *minimal progress* in a history H if, in every suffix of H, *some pending active invocation* has a matching response. In other words, there is no point in the history where *all threads that called the method* take an infinite number of concrete steps without returning. A method provides *maximal progress* in a history H if, in every suffix of H, *every pending active invocation* has a matching response. In other words, there is no point in the history where *a thread that called the method* takes an infinite number of concrete steps without returning. Following these definitions, the wait-free and lock-free data structures are mapped in the periodical table as scheduler independently providing maximal and minimal progress, respectively.

Historically, a number of so-called *universal methods* for constructing non-blocking data structures of any type have been discussed in the literature [15, 18, 19, 20]. The first correct CAS-

based lock-free list-based set proposal was introduced by Harris [21]. Later, Michael improved Harris work by presenting a proposal that was compatible with all lock-free memory management methods and Michael used this proposal as a building block for lock-free hash maps [22]. The proposal allowed the concurrent search, insert and remove operations in a lock-free manner, but the size of the arrays of buckets was fixed and expanding was typically implemented with a global lock. Shalev and Shavit extended Michael's work when they presented their lock-free algorithm for expanding hash maps [23]. The algorithm is based on split-ordered lists and allows the number of hash buckets to vary dynamically according to the number of nodes inserted or removed, preserving the read-parallelism. More recently, Triplett *et al.* presented a set of algorithms that allow concurrent wait-free linear scalable searches while shrinking and expanding hash maps [24].

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. Skip-lists were originally invented by Pugh [25]. Concurrent non-blocking skip-lists were later implemented by Herlihy *et al.* [26]. Regarding concurrent hash trie data structures, recently Prokopec *et al.* presented the *CTries* [14], 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.

3. Our Proposal By Example

In a nutshell, our design has *hash arrays of buckets* and *leaf nodes*. The leaf nodes store key/content pairs and the hash arrays of buckets implement a hierarchy of hash levels of fixed size 2^w. To map a key/content pair (k, c) into this hierarchy, we first compute the hash value h for k and then use chunks of w bits from h to index the entry in the appropriate hash level, i.e., for each hash level H_i, we use the ith group of w bits of h to index the entry in the appropriate bucket array of H_i. Hash collisions are solved by simply walking down the tree as we consume successive chunks of w bits from the hash value h, creating a unique path from the root level of the hash to the level where (k, c) should be stored. In what follows, we discuss the key aspects of our proposal. We begin with Fig. 2 showing a small example that illustrates how the concurrent insertion of nodes is done in a hash level.

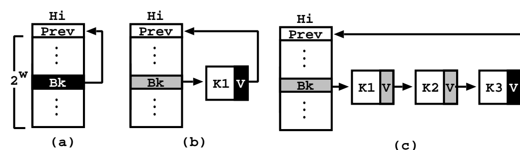

Figure 2: Insert operation in a hash level

Figure 2(a) shows the initial configuration for a hash level. Each hash level H_i is formed by a bucket array of 2^w entries and by a backward reference to the previous hash level (represented as $Prev$ in the figures). For the root level, the backward reference is $Null$. In Fig. 2(a), B_k represents a particular bucket entry of the hash level. B_k and the remaining entries are all initialized with a reference to the current level H_i. During execution, each bucket entry stores either a reference to a hash level or a reference to a separate chaining mechanism, using a chain of internal nodes, that deals with the hash collisions for that entry. Each internal node holds a key/content pair (for the sake of simplicity of presentation, we only show the keys in the figures) and a tuple that holds both a reference to a next-on-chain internal node and the condition of the node, which can be valid (V) or invalid (I). The initial condition of a node is valid (V). Figure 2(b) shows the hash configuration after the insertion of node K_1 on the bucket entry B_k and Fig. 2(c) shows the hash configuration after the insertion of nodes K_2 and K_3 also in B_k. Note that the insertion of new nodes is done at the end of the chain and any new node being inserted closes the chain by referencing back the current level.

During execution, the memory locations holding references are considered to be in one of the following states: *black*, *white* or *gray*. A black state, which we also name an *Interest Point (IP)*, represents a memory location that will be used to update the state of a chain or a hash level in a concurrent fashion. To guarantee the property of lock-freedom, all updates to black states are done using CAS operations. A gray state represents a memory location that is not an IP but which can become an IP at any instant, once the execution leads to it.

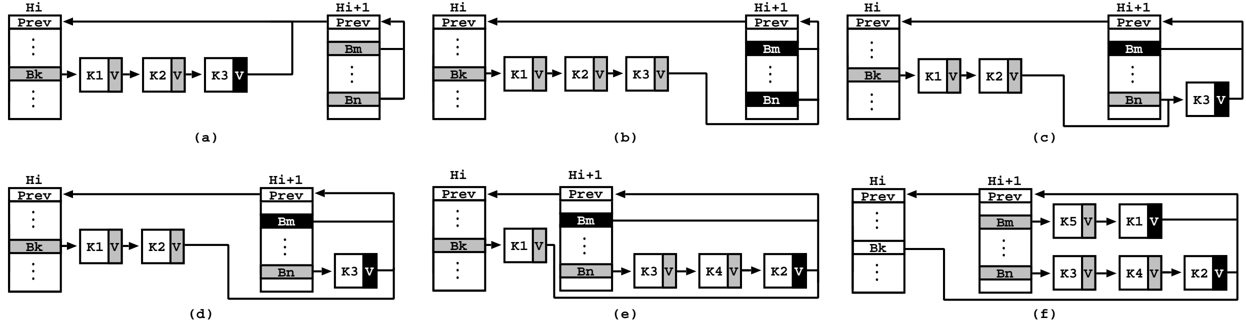

Figure 3: Expand operation from a bucket entry to a second level hash with concurrent insertion of nodes

A white state represents a memory location used only for reading purposes. As the hash trie evolves during time, a memory location can change between black and gray states until reaching the white state, where it is no longer updated.

When the number of valid nodes in a chain exceeds a threshold value MAX_NODES, then the corresponding bucket entry is expanded with a new hash level and the nodes in the chain are remapped in the new level. Thus, instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w.

Starting from the configuration in Fig. 2(c), Fig. 3 illustrates the expansion mechanism with a second level hash for the bucket entry B_k. The expansion operation is activated whenever a thread T meets the following two conditions: (i) the key at hand was not found in the chain and (ii) the number of valid nodes in the chain observed by T is equal to the threshold value (in what follows, we consider a threshold value of three nodes). In such case, T starts by pre-allocating a second level hash H_{i+1}, with all entries referring the respective level (Fig. 3(a)). The new hash level is then used to implement a synchronization point with the current IP (node K_3 in Fig. 3(a)) that will correspond to a successful CAS operation trying to update H_i to H_{i+1} (Fig. 3(b)). From this point on, the insertion of new nodes on B_k will be done starting from the new hash level H_{i+1}.

If the CAS operation fails, that means that another thread has gained access to the IP and, in such case, T aborts its expansion operation. Otherwise, T starts the remapping process of placing the internal valid nodes K_1, K_2 and K_3 in the correct bucket entries in the new level. Figures 3(c) to 3(f) show the remapping sequence in detail. For simplicity of illustration, we will consider only the entries B_m and B_n on level H_{i+1} and assume that K_1, K_2 and K_3 will be remapped to B_m, B_n and B_n, respectively. 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 new 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.

Figure 3(c) shows the hash trie configuration after the successful CAS operation that adjusted node K_3 to entry B_n. After this step, B_n passes to the gray state and K_3 becomes the next IP for the insertion of new nodes on B_n. Note that the initial chain for B_k has not been affected yet, since K_2 still refers to K_3. Next, on Fig. 3(d), the chain is adjusted and K_2 is updated to refer to the second level hash H_{i+1}. The process then repeats for K_2 (the new last node on the chain for B_k). First, K_2 is remapped to entry B_n and then it is removed from the original chain, meaning that the previous node K_1 is updated to refer to H_{i+1} (Fig. 3(e)). Finally, the same idea applies to K_1. In the continuation, K_1 is also remapped to a bucket entry on H_{i+1} (B_m in the figure) and then removed from the original chain, meaning in this case that the bucket entry B_k becomes itself a reference to H_{i+1} (Fig. 3(f)). From now on, B_k is a white memory location since it will be no further updated. 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. 3(e), where a node K_4 is inserted before K_2 in B_n, and in Fig. 3(f), where a node K_5 is inserted before K_1 in B_m.

We now move to the description of the remove operation. In our proposal, a remove operation can be seen as a sequence of two steps: (i) the *invalidation*

step; and (ii) the *invisibility step*. The invalidation step searches for the node N holding the key to be removed and updates the node condition from valid to invalid. The invisibility 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. 2(c), where the initial condition of all keys is valid, Fig. 4 illustrates how the concurrent removal of nodes is done.

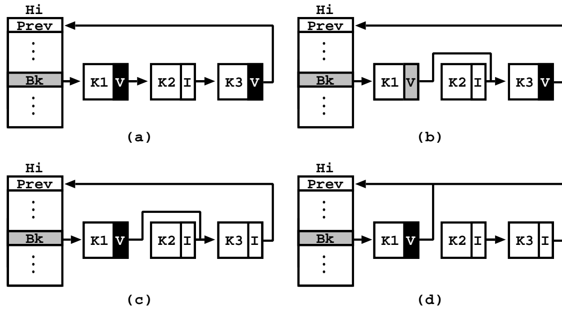

Figure 4: Remove operation in a hash level

Assume now 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, which in turn makes node K_1 into a second IP (Fig. 4(a)). In the continuation, T searches for the previous/next valid data structure before/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 (invisibility step).

Figure 4(c) shows the remove operation for key K_3. The node for K_3 is first marked as invalid, which in turn makes again node K_1 an IP (K_1 is the previous valid data structure before K_3). Since K_3 is the last node in the chain for bucket B_k, the next valid structure after K_3 is H_i. Thus, in the invisibility step, K_3 is bypassed by updating K_1 to refer to H_i (Fig. 4(d)). The reader can observe that, at this point, nodes K_2 and K_3 are not in the chain. However, their chaining references are left in a consistent state, allowing all late threads reading those nodes to be able to recover to a valid data structure, which in Fig. 4(d) is the hash level H_i.

Next, Fig. 5 starts from the hash trie configuration shown in Fig. 4(c) and presents a situation where two threads are inserting and removing keys simultaneously in the same chain. Assume that thread T is removing the keys K_3 and K_1 and thread U is inserting the keys K_4 and K_5. As the lock-freedom property must hold in any situation, thread U must be able to insert K_4 and K_5 without waiting for any other thread.

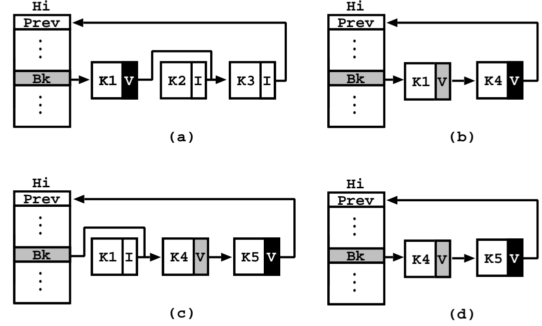

Figure 5: Remove operation with concurrent insertion of nodes

Note that, with the remove operation, the last node in the chain is not necessarily a valid node, which in turn does not guarantee that an IP is always referring back the current hash level. This is the case of nodes K_1 and K_3 in Fig. 5(a). Node K_1 is the current IP but since it is referring the invalid node K_3, it is not the last node in the chain. On the other hand, node K_3 is an invalid node but since it is the last node in the chain, it is referring back the current hash level H_i. Figure 5(b) then shows the insertion of node K_4 and the change on the IP from K_1 to K_4 (for the sake of simplicity, nodes K_2 and K_3 are no longer shown). Next, Fig. 5(c) shows the simultaneous insertion and removal of nodes K_5 and K_1, respectively. Finally, Fig. 5(d) shows the final chain of nodes at the end of both operations.

We conclude the presentation of our proposal with Fig. 6 showing a last situation where a thread T is executing the expand operation (like in the example of Fig. 3) and another thread U is removing a key from the nodes being adjusted. Figure 6(a) shows the initial configuration where T already connected the last node in the chain to the new hash level H_{i+1} and prepares itself to start the remapping process of placing the internal nodes K_3, K_2 and K_1 in the correct bucket entries of H_{i+1}. Concurrently, thread U is removing key K_3 and has already marked as invalid the node K_3 in the invalidation step. In the invisibility step, thread U is led to the hash level H_{i+1} by following the chain reference of K_3 but, since K_3 has not yet been adjusted, thread U is not able to find K_3 in H_{i+1}. Thread U knows that this situation is only possible because another thread is simultaneously executing

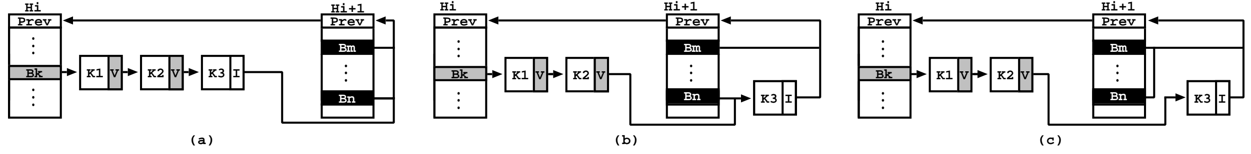

Figure 6: Expand operation from a bucket entry to a second level hash with concurrent removal of nodes

an expand operation. In such scenario, thread U delegates the completion of the unfinished removal operations to the thread doing the expansion and finishes the execution of its remove operation.

Since initially T saw K_3 as valid, in the continuation, T adjusts K_3 to the bucket entry B_n (Fig. 6(b)). To overcome the fact that K_3 is marked as invalid, now T also needs to check if a node remains valid after being adjusted and, if not, T executes the invisibility step for the node (Fig. 6(c)). Node K_3 is thus bypassed in the hash level H_{i+1}, but it is still chained to K_2. This is not a problem because, in the continuation of the adjustment process, it will be also bypassed with node K_2 being chained to H_{i+1}.

4. Algorithms

This section discusses in more detail the key algorithms that implement our proposal. We begin with Alg. 1 which shows the pseudo-code for the process of adjusting a given node N into a given hash level H. The algorithm begins by updating the chain reference (field *NextRef()*) of N to H using a $ForceCAS()$ procedure, which repeats a CAS operation until it succeeds. Next, since only valid nodes need to be adjusted, it checks if N is a valid node (invalid nodes are left unchanged) and applies the hash function that allows obtaining the bucket entry B in H that fits the key on N (line 4).

In the continuation, if B is empty (lines 5–9), the algorithm tries to insert N on the head of B by using a CAS operation (line 6). If successful, then it checks if, in the meantime, N turned invalid (this situation corresponds to the scenario described in Fig. 6 where the thread invalidating N delegates the unfinished removal operations to the thread doing the expansion), and if so, it calls the *MakeNodeInvisible()* procedure (see Alg. 4 next) to remove the node from the chain (line 8).

If B is not empty, the algorithm then checks whether the head reference R in B refers a second hash level, case in which it calls itself (lines 10–12). Otherwise, it starts traversing the chain of nodes searching for valid candidate nodes (lines 13–22). For that, it uses three auxiliary variables: CR holds the candidate reference where new insertions/adjustments should take place; CRN holds the chain reference of CR; and C counts the number of valid nodes in the chain. Initially, CR is the bucket entry from where the chain starts, R and CRN are the head node in the chain and C is 0. At the end, the algorithm checks if R ended in the same hash level H, which means that no other expansion operation is taking place at the same time, and it proceeds trying to adjust N (lines 24–47). Otherwise, R refers a deeper hash level, case in which the algorithm is restarted in the hash level after H, using the $Prev$ backward references to move up in the hash levels (lines 48–49). If R ended in the same hash level then two situations might occur: no valid chain nodes were found (lines 25–30) or, at least, a valid node was found (lines 32–45).

If no valid nodes were found then the algorithm tries to insert N in the head of the chain (line 25) and, if the CAS succeeds, it checks if, in the meantime, N turned invalid and must be removed (line 27). Otherwise, if the CAS fails, that means that another thread has updated the head of the bucket entry in the meantime. In such case, the algorithm reads the new head reference R (line 30) and the process is restarted in the same hash level H (lines 46–47) or in the hash level after H (line 48–49), case R references a deeper hash level.

If a valid node was found (lines 32–45) then the algorithm tries to insert N after the CR node (line 40). As before, we follow the same steps case the CAS operation succeeds (lines 41–43) or fails (line 45). Please note that here, for the CAS operation, we use the *Next()* field as a way to represent simultaneously the pair holding the chain reference and the condition of the node. This is necessary since we need to guarantee that the node is valid when updating a chain reference.[1]

[1]At the implementation level, the chain reference and the condition of the node are, in fact, treated as a single field.

Algorithm 1 *AdjustNode(Node N, Hash H)*

```
 1: ForceCAS(NextRef(N), H)
 2: if IsInvalidNode(N)
 3:   return
 4: B ← GetHashBucket(H, Hash(Level(H), Key(N)))
 5: if EntryRef(B) = H    // B is an empty bucket
 6:   if CAS(EntryRef(B), H, N)
 7:     if IsInvalidNode(N)
 8:       MakeNodeInvisible(N, H)
 9:     return
10: R ← EntryRef(B)
11: if IsHash(R)    // R references a second hash level
12:   return AdjustNode(N, R)
13: CR ← B
14: CRN ← R
15: C ← 0
16: repeat   // traverse the chain of nodes
17:   if IsValidNode(R)
18:     CR ← R
19:     CRN ← NextRef(CR)
20:     C ← C + 1
21:   R ← NextRef(R)
22: until IsHash(R)
23: if R = H    // chain ended in the same hash level
24:   if C = 0    // no valid chain nodes found
25:     if CAS(EntryRef(B), CRN, N)
26:       if IsInvalidNode(N)
27:         MakeNodeInvisible(N, H)
28:       return
29:     else
30:       R ← EntryRef(B)
31:   else   // a valid node was found
32:     if C = MAX_NODES    // chain is full
33:       newH ← AllocInitHash(Level(H) + 1)
34:       PrevHash(newH) ← H
35:       if CAS(Next(CR), (CRN, valid), (newH, valid))
36:         ExpandToNewHash(newH, N, H)
37:         return AdjustNode(N, newH)
38:       else
39:         FreeHash(newH)
40:     if CAS(Next(CR), (CRN, valid), (N, valid))
41:       if IsInvalidNode(N)
42:         MakeNodeInvisible(N, H)
43:       return
44:     else
45:       R ← NextRef(CR)
46:   if IsNode(R)
47:     return AdjustNode(N, H)
48: R ← GetPrevHash(R, Level(H) + 1)
49: return AdjustNode(N, R)
```

Before trying to insert N after the CR node, the algorithm needs to check if the chain is full (line 32), in which case it starts a new expand operation (lines 33–39). Here, a new hash level $newH$ is first allocated and initialized (lines 33–34) and then used to implement a synchronization point that will correspond to a CAS operation trying to update the chain reference in CR from CRN to $newH$ (line 35). If the CAS succeeds, the algorithm gains access to the expand operation and starts the remapping process of placing the valid nodes in the chain to the new hash level $newH$ (line 36) and, at the end, *AdjustNode()* is called again, this time for the new hash level (line 37). Otherwise, if the CAS fails, that means that another thread gained access to the expand operation and, in such case, the algorithm aborts the new expand operation, frees $newH$ (line 39) and continues.

Next, we present the procedure that supports the remove operation. Algorithm 2 shows the pseudo-code for the search/remove operation of a given key K in a given hash level H. The algorithm begins by applying the hash function that allows obtaining the bucket entry B of H that fits K (line 1). In the continuation, if B is empty then K was not found and the algorithm finishes (lines 2–3). If B is not empty, it checks whether the head reference R in B refers a second hash level, case in which it calls itself (line 6).

Algorithm 2 *SearchRemoveKey(Key K, Hash H)*

```
 1: B ← GetHashBucket(H, Hash(Level(H), K))
 2: if EntryRef(B) = H    // B is an empty bucket
 3:   return
 4: R ← EntryRef(B)
 5: if IsHash(R)    // R references a second hash level
 6:   return SearchRemoveKey(K, R)
 7: CR ← B
 8: CRN ← R
 9: repeat   // traverse the chain of nodes
10:   if IsValidNode(R)
11:     if Key(R) = K    // found key in R
12:       if MakeNodeInvalid(R)
13:         return MakeNodeInvisible(R, H)
14:     else
15:       CR ← R
16:       CRN ← NextRef(CR)
17:   R ← NextRef(R)
18: until IsHash(R)
19: if R = H    // chain ended in the same hash level
20:   return
21: R ← GetPrevHash(R, Level(H) + 1)
22: return SearchRemoveKey(K, R)
```

Otherwise, it starts traversing the chain of nodes searching for a valid node R with K (lines 7–18).

If K is found (line 11) then it tries to mark R as invalid and, if successful (i.e., R was valid and the call to *MakeNodeInvalid()* turned R invalid), it proceeds to the invisibility step by calling *MakeNodeInvisible()* and returns (lines 12–13). Otherwise, if *MakeNodeInvalid()* fails (i.e., R was already marked as invalid by another thread), the algorithm continues in search mode. During search, the CR and CRN references are updated as discussed in Alg. 1 until R reaches a hash level (line 18). If R ends in the same hash level H, that means that no expansion operation is taking place at the same time, and the algorithm simply returns (lines 19–20). Otherwise, R refers a deeper hash level, case in which the algorithm is restarted in the hash level after H (lines 21–22).

Algorithm 3 presents the process of invalidating a given node N. This process is used in the remove operation to force a change in the state of a node from valid to invalid. The algorithm applies repeatedly a CAS operation with an invalid state flag over the node until the CAS either succeeds (thus changing the state of N from valid to invalid) or until it detects that N was already marked as invalid by another thread.

Algorithm 3 *MakeNodeInvalid(Node N)*

```
1: repeat
2:   R ← NextRef(N)
3:   if IsInvalidNode(N)    // node is already invalid
4:     return False
5: until CAS(Next(N), (R, valid), (R, invalid))
6: return True
```

Finally, Alg. 4 presents the pseudo-code for turning invisible a given node N in a given hash level H. Remember that, in the invisibility step, we need to search for the valid data structures before and after N in the chain of nodes, respectively BR (*before reference*) and AR (*after reference*) in Alg. 4, 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 it moves until the hash at the end of the chain (line 4). Next, if the algorithm ended in the same hash level H (line 5), 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).

Algorithm 4 *MakeNodeInvisible(Node N, Hash H)*

```
 1: R ← GetNextHashOrValidNode(N)
 2: AR ← R
 3: if IsNode(R)
 4:   R ← GetNextHash(R)
 5: if R = H    // chain ended in the same hash level
 6:   B ← GetHashBucket(H, Hash(Level(H), Key(N)))
 7:   R ← B
 8:   repeat
 9:     BR ← R
10:     BRN ← NextRef(BR)
11:     R ← GetNextHashOrValidNodeOrN(R)
12:   until R = N ∨ IsHash(R)
13:   if R = H    // N is already invisible
14:     return
15:   if R = N    // we are in condition to bypass N
16:     if BR = B    // no valid chain nodes found
17:       if CAS(EntryRef(BR), BRN, AR)
18:         return
19:     else
20:       if CAS(Next(BR), (BRN, valid), (AR, valid))
21:         return
22:     return MakeNodeInvisible(N, H)
23: return MakeNodeInvisible(N, R)
```

At the end of the traversal, if R is H that means that N is already invisible, thus the algorithm simply returns (lines 13–14). If R is N that means that we are in condition to bypass N by chaining BR to AR (lines 15–22). 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 condition as valid if BR is a node (line 20). Notice that if the CAS operation fails, then it means that the BR node has been update 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 chain configuration where all invalid nodes are made invisible.

Otherwise, if R ends in a hash level at the end of the traversal, that means that N is not on H. Therefore, R refers to a deeper hash level and the process is restarted in that hash level (line 23).

5. Correctness & Complexity

In this section, we discuss the correctness and complexity of our proposal. The full correctness proof consists in two parts: first prove that the proposal is *linearizable* and then prove that *progress*

occurs in a lock-free fashion for all operations.

5.1. Linearizability

Linearizability is an important correctness condition for the implementation of concurrent data structures [27]. An operation is linearizable if it appears to the rest of the system to take effect instantaneously at some moment of time between its invocation and its response. Linearizability guarantees that if all operations individually preserve an invariant, the system as a whole also will. In what follows, we focus on the linearization proof and we describe the linearization points of the proposal, the set of invariants, and the proof that the linearization points preserve the set of invariants.

The linearization points in the algorithms shown are:

$\boldsymbol{LP_1}$ *AdjustNode()* (Alg. 1) is linearizable at the *ForceCAS()* in line 1.

$\boldsymbol{LP_2}$ *AdjustNode()* (Alg. 1) is linearizable at the *CAS()* in line 6.

$\boldsymbol{LP_3}$ *AdjustNode()* (Alg. 1) is linearizable at the *CAS()* in line 25.

$\boldsymbol{LP_4}$ *AdjustNode()* (Alg. 1) is linearizable at the *CAS()* in line 35.

$\boldsymbol{LP_5}$ *AdjustNode()* (Alg. 1) is linearizable at the *CAS()* in line 40.

$\boldsymbol{LP_6}$ *MakeNodeInvalid()* (Alg. 3) is linearizable at the *CAS()* in line 5.

$\boldsymbol{LP_7}$ *MakeNodeInvisible()* (Alg. 4) is linearizable at the *CAS()* in line 17.

$\boldsymbol{LP_8}$ *MakeNodeInvisible()* (Alg. 4) is linearizable at the *CAS()* in line 20.

The set of invariants that must be preserved are:

$\boldsymbol{Inv_1}$ For every hash level H, $PrevHash(H)$ always refers to the previous hash level.

$\boldsymbol{Inv_2}$ 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 node N_1; (iii) after a follower update, it must refer either to another node N_2, to the hash level H or to a second hash level H_d such that $PrevHash(H_d) = H$. If B is referring to H_d, then no more updates occur in B.

$\boldsymbol{Inv_3}$ A node N must comply with the following semantics: (i) its initial condition is *valid*; (ii) after an update to *invalid*, it never changes again to *valid*.

$\boldsymbol{Inv_4}$ A node N must comply with the following semantics: (i) valid nodes are always *visible* to all threads; (ii) invalid nodes can be *temporarily visible* to some threads before made *invisible* to all threads.

$\boldsymbol{Inv_5}$ A valid node N in a chain of nodes starting from a bucket entry B belonging to a hash level H must comply with the following semantics: (i) its initial (next-on-chain) reference is H; (ii) after an update, it must refer to another node in the chain or to a hash level; (iii) once it refers to a hash level H_d (at least one level) deeper than H, then it never refers again to H.

$\boldsymbol{Inv_6}$ An invalid node N in a chain of nodes starting from a bucket entry B belonging to a hash level H must comply with the following semantics: (i) if N is temporarily visible then its (next-on-chain) reference can be updated to a hash level H_d (at least one level) deeper than H; (ii) if N is invisible then its reference is never updated again.

$\boldsymbol{Inv_7}$ The number of valid nodes in a chain of nodes starting from a bucket entry B belonging to a hash level H is always lower or equal than a predefined threshold value $MAX_NODES \geq 1$.

$\boldsymbol{Inv_8}$ Given a key K not present in the hash map, it exists only one path P from the root hash level to the insertion point IP (a hash bucket or a chain node), where the node with K should be inserted. At any given instant, the insertion point IP is unique and is always the last valid data structure in the path P.

$\boldsymbol{Inv_9}$ Given a key K present in the hash map, it exists only one path P from the root hash level to the invisibility point IP (a hash bucket or a valid chain node), from where the node N with K should be made invisible. After N be marked as invalid, the invisibility point IP is unique.

Next, we show the proof strategy used to prove that the linearization points preserve the set of invariants. For simplicity of presentation, we show

the proof only for the linearization points LP_1 and LP_2. The remaining linearization points follow a similar proof strategy.

Lemma 1. *In the initial state of the data structure the set of invariants holds.*

PROOF. Consider that H represents the root level for a hash trie (its initial configuration is the same as the one represented in Fig. 2(a)). Since H is the root level, the reference $PrevHash(H)$ is $Null$ (Inv_1), each bucket entry B is referring H (Inv_2) and the number C of nodes in any chain is 0 (Inv_7). The remaining invariants are not affected. □

Lemma 2. *The linearization point LP_1 preserves the set of invariants.*

PROOF. After the execution of the *ForceCAS()* procedure in line 1, node N refers to the hash level H and its state remains unchanged, thus invariants Inv_3, Inv_4, Inv_5, Inv_6 and Inv_7 hold. The remaining invariants are not affected. □

Lemma 3. *The linearization point LP_2 preserves the set of invariants.*

PROOF. After the successful execution of the *CAS()* procedure at line 6, the bucket entry B refers to N (Inv_2), N refers to H (Lemma 2) and the number of valid nodes in the chain is $C = 1$ (Inv_7). Inv_4 holds because if N is valid then it is visible, otherwise if N is invalid then it is temporarily visible but will become invisible after the execution of the *MakeNodeInvisible()* procedure in line 8. The remaining invariants are not affected. □

5.2. Progress

To prove that progress occurs in a lock-free fashion for all operations, we begin by defining the three types of stages that specify the type of progress a thread can make:

- the private stages (represented by white double rectangles in what follows) do not change the configuration of the data structures. A thread progresses in a private fashion when traversing the data structures without changing them.
- the public stages where a thread *might* change the configuration of the data structures (gray rectangles). Examples are the insertion of a new key, the removal of an existent key or making a node invisible.
- The public stages where a thread *must* change the configuration of data structures (black rectangles). Examples are the operations required for hash expansion.

Figure 7 shows the progress stages of a thread for the search/insert operation and Fig. 8 shows the progress stages of a thread for the search/remove operation (in both figures, the oval boxes represent the decisions points of the algorithms).

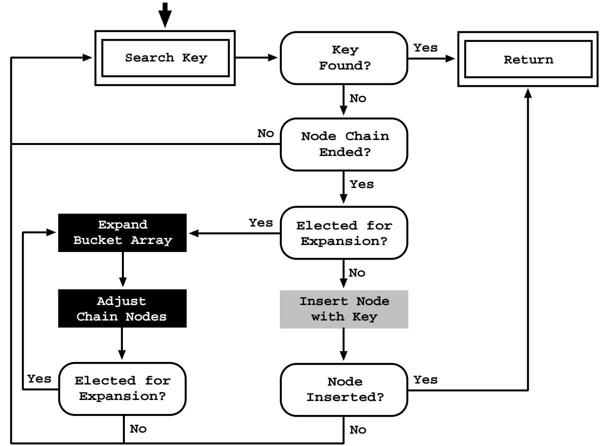

Figure 7: Progress stages for the search/insert operation

For the search/insert operation, the entry point is the *Search Key* stage. In this stage, the thread searches for a key in the chain of nodes and if the key is found then the thread passes to the *Return* stage. Otherwise, if the key is not found in the chain of nodes, the thread tries to insert it. Before that, if the conditions for expansion hold, it can be elected to execute expansion. If elected, the thread passes to the *Expand Bucket Array* stage, where first it expands the current bucket array and then adjusts all nodes in it to the new hash level, the *Adjust Chain Nodes* stage. At the end of this process, it can be elected for another expansion, otherwise it returns to the *Search Key* stage. If not elected to execute expansion, it moves to the *Insert Node with Key* stage. Here, if the node is not inserted, it moves again to the *Search Key* stage. Otherwise, the node was successfully inserted and the thread passes to the *Return* stage.

For the search/remove operation, the entry point is also the *Search Key* stage, where the thread searches for a key in the chain of nodes and if the key is found then the thread passes to the *Make Node Invalid* stage. In this stage, if the thread succeeds in turning the node invalid, it executes the

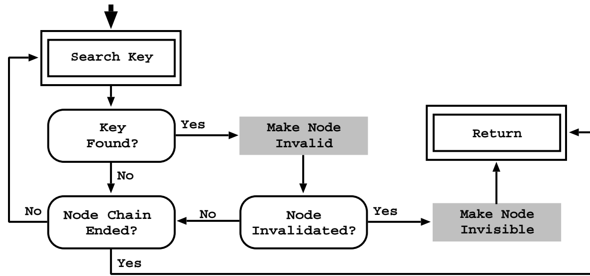

Figure 8: Progress stages for the search/remove operation

Make Node Invisible stage and returns. Otherwise, if the thread fails in turning the node invalid, it continues assuming that it has not found the key and passes again to the *Search Key* stage or returns if reaching the end of the chain.

5.3. Complexity

Our design implements a hierarchy of hash levels whose branching factor is given by a fixed (and predefined) 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. 2, we showed three keys associated to a single bucket. However, for the sake of simplicity, next we will formalize the complexity of our design, using 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 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 space complexity on the number of hashes used is $\mathcal{O}(\frac{K}{E})$.*

Proof. In a worst case scenario, one would have key collisions up to the leaf hash level, where the keys must be necessarily different, otherwise they would be the same and our design does not store repeated keys.

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 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 5. *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 used is $\mathcal{O}(\log_E K)$.*

Proof. In a worst case scenario, one would have key collisions up to the leaf hash level, where the keys must be necessarily different, otherwise they would be the same and our design does not store repeated keys.

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 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 proposal with other state-of-the-art concurrent hash map designs. The environment for our experiments was a SMP (Symmetric Multi-Processing) system, based in a NUMA (Non-Uniform Memory Access) architecture with 32-Core AMD Opteron Processor 6274 (2 sockets with 16 cores each) with 32GB of main memory, each processor with caches L1, L2 and L3 respectively with 64KB, 2048KB and 6144KB, running the Linux kernel 3.18.6-100.fc20.x86_64 with Oracle's Java Development Kit 1.8.0_66.

Although our proposal 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 invisible/unreachable data structures; and (ii) easy comparison against other hash map designs. Some of the best-known hash map implementations currently available are already implemented in the Java library, such as

the *Concurrent Hash Maps (CHM)* and the *Concurrent Skip-Lists (CSL)* from the Java's concurrency package. Additionally, we will be comparing our proposal against Click's *Non Blocking Hash Maps (NBHM)* [13] and Prokopec *et al.* *Concurrent Tries (CT)* [14].[2] We have ran our proposal with a MAX_NODES threshold value of 6 chain nodes for the hash collisions and with two different configurations for the number of buckets entries per hash level, one with 8 and another with 32 buckets entries per hash level. In what follows, we will name our proposal as *Free Fixed Persistent Hash Map (FFP)* and those two configurations as FFP_8 and FFP_{32}, respectively. To put the five proposals in perspective, Table 1 shows how they support/implement the features of (i) be lock-freedom; (ii) use fixed size data structures; and (iii) maintain the access to all internal data structures as persistent memory references.

Table 1: Features supported by the proposals evaluated

Features / Proposals	CHM	CSL	NBHM	CT	FFP
Lock-freedom	✗	✗	✓	✓	✓
Fixed size structures	✗	-	✗	✓	✓
Persistent references	✗	✓	✓	✗	✓

To test the proposals, we developed a testing environment that includes benchmark sets with different percentages of insert, search and remove operations, for a fixed size of 10^6 randomized items.[3] To spread threads among a benchmark set S, we divide the size of S by the number of running threads and place each thread in a position within S in such a way that all threads perform the same number of inserts, searches and removes on S. For the search and remove operations, the corresponding items are inserted beforehand and without counting to the execution time. To warm up the Java Virtual Machine, we ran each benchmark 5 times beforehand and then we took the average execution time of the next 20 runs.

Table 2 shows the execution time results[4] obtained for the *CHM*, *CSL*, *NBHM*, *CT*, FFP_8 and FFP_{32} proposals using six benchmark sets that vary in the percentage of concurrent operations to be executed. The 1st benchmark only performs inserts, the 2nd only searches, and the 3rd only removes. The remaining benchmarks perform mixed operations with different percentages of inserts, searches and removes. For each benchmark, Table 2 shows the execution time, in milliseconds, and speedup ratio for 1, 8, 16, 24 and 32 threads.

Analyzing the general picture of the table, one can observe that, for these benchmarks, each proposal has it own advantages and disadvantages, i.e., there is no single proposal that overcomes all the remaining proposals. For the execution times, the table shows a clear trade-off balance between the concurrent insertion, search and removal of items. The proposals with the best execution times in the concurrent insertions are not so good in the concurrent searches and the same happens with the concurrent removal of items.

When the weight of insertions is high, as in the 1st and 4th benchmarks, our proposal outperforms the remaining proposals. Clearly, the FFP_{32} proposal has the best base times (one thread) and, as we increase the number of threads, both FFP_8 and FFP_{32} proposals are able to scale properly. In particular, for 32 threads, FFP_{32} achieves the best execution times. We explain the performance of our proposal with the trie design and the hash function that spreads potential synchronization points among the trie, minimizing this way false-sharing and cache ping-pong effects. The FFP_{32} has better results than FFP_8 because it expands hash levels more aggressively, i.e., on each expansion it consumes 5 bits of the hash key, while FFP_8 only consumes 3 bits, thus reducing hash collisions of keys in a hash level.

On the other hand, when the weight of search operations is high, as in the 2nd and 5th benchmarks, our proposal is not as efficient as the other proposals. In the 2nd benchmark, the *CHM* proposal shows the best base times, while *NBHM* shows the best results as we increase the number of threads. In the 5th benchmark, *CHM* has the best execution times and our FFP_{32} proposal is the second best. In order to understand why our proposal is not so good in the search operation, we measured the time that threads spent just in the hash trie

[2] Both downloaded on January 18, 2016 from `https://github.com/boundary/high-scale-lib` and `https://github.com/romix/java-concurrent-hash-trie-map/tree/master/src/main/java/com/romix/scala/collection/concurrent`, respectively.

[3] Available from `https://github.com/miar/ffp`

[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.

Table 2: Execution time, in milliseconds, for the execution with 1, 8, 16, 24 and 32 threads and the corresponding speedup ratios against 1 thread, for six benchmark sets using different ratios for the number of concurrent insert, search and remove operations (for each configuration, the best execution times and speedups are in bold)

# Threads	**Execution Time** (E_{T_p})						**Speedup Ratio** (E_{T_1}/E_{T_p})					
(T_p)	**CHM**	**CSL**	**NBHM**	**CT**	**FFP$_8$**	**FFP$_{32}$**	**CHM**	**CSL**	**NBHM**	**CT**	**FFP$_8$**	**FFP$_{32}$**
1st – Insert: 100% Search: 0% Remove: 0%												
1	663	3,238	12,968	919	946	**542**						
8	294	550	2,933	207	**174**	176	2.26	**5.89**	4.42	4.44	5.44	3.08
16	199	332	2,031	118	**117**	124	3.33	**9.75**	6.39	7.79	8.09	4.37
24	201	276	1,717	107	**96**	153	3.30	**11.73**	7.55	8.59	9.85	3.54
32	212	270	1,576	97	89	**74**	3.13	**11.99**	8.23	9.47	10.63	7.32
2nd – Insert: 0% Search: 100% Remove: 0%												
1	**155**	3,753	225	773	720	379						
8	38	535	**34**	120	118	76	4.08	**7.01**	6.62	6.44	6.10	4.99
16	27	327	**25**	78	76	53	5.74	**11.48**	9.00	9.91	9.47	7.15
24	30	309	**22**	70	64	53	5.17	**12.15**	10.23	11.04	11.25	7.15
32	32	315	**26**	78	69	54	4.84	**11.91**	8.65	9.91	10.43	7.02
3rd – Insert: 0% Search: 0% Remove: 100%												
1	**314**	4,144	451	1,585	872	582						
8	**105**	595	122	226	172	137	2.99	6.96	3.70	**7.01**	5.07	4.25
16	**62**	341	77	156	108	89	5.06	**12.15**	5.86	10.16	8.07	6.54
24	**55**	303	66	132	94	130	5.71	**13.68**	6.83	12.01	9.28	4.48
32	**54**	306	64	124	101	102	5.81	**13.54**	7.05	12.78	8.63	5.71
4th – Insert: 60% Search: 30% Remove: 10%												
1	721	2,510	15,342	1,027	873	**618**						
8	150	413	4,030	174	148	**142**	4.81	**6.08**	3.81	5.90	5.90	4.35
16	128	247	2,803	115	**91**	106	5.63	**10.16**	5.47	8.93	9.59	5.83
24	75	191	2,566	89	**72**	74	9.61	**13.14**	5.98	11.54	12.13	8.35
32	72	178	1,870	90	80	**67**	10.01	**14.10**	8.20	11.41	10.91	9.22
5th – Insert: 20% Search: 70% Remove: 10%												
1	**282**	1,890	12,370	764	757	395						
8	**51**	282	8,517	171	157	74	5.53	**6.70**	1.45	4.47	4.82	5.34
16	**39**	184	3,623	87	72	82	7.23	10.27	3.41	8.78	**10.51**	4.82
24	**37**	143	3,058	73	69	64	7.62	**13.22**	4.05	10.47	10.97	6.17
32	**38**	145	2,081	74	69	65	7.42	**13.03**	5.94	10.32	10.97	6.08
6th – Insert: 25% Search: 50% Remove: 25%												
1	**279**	2,059	12,181	1,087	808	440						
8	113	340	3,125	159	127	**83**	2.47	6.06	3.90	**6.84**	6.36	5.30
16	**64**	214	3,482	104	82	70	4.36	9.62	3.50	**10.45**	9.85	6.29
24	**42**	180	2,609	87	71	78	6.64	11.44	4.67	**12.49**	11.38	5.64
32	**44**	166	1,902	83	77	66	6.34	12.40	6.40	**13.10**	10.49	6.67

levels for the FFP_8 and FFP_{32} proposals and we noticed that, if we subtracted such time to the overall execution time, we got execution times similar to those of *CHM*. Notice that the *CHM* proposal implements a single hash level, which is expanded each time the hash becomes saturated, and for hash collisions in bucket entries, it uses a separate chaining mechanism that is implemented as a red-black tree whenever the chain becomes saturated.

A further profiling study lead us to conclude that our proposal is actually suffering from a cache miss penalty when threads navigate through many hash levels. We took some internal statistics about the depth of the hash levels used on both FFP_8 and FFP_{32} configurations. For example, for the 2nd benchmark, the FFP_8 configuration has a minimum and a maximum hash trie depth of 5 and 7, respectively, and an average number of nodes in non-empty chains of 2.39. The FFP_{32} configuration has a minimum and a maximum depth of 4 and 6, respectively, and an average number of nodes in non-empty chains of 1.48. Thus, the higher the number of bucket entries per hash level, the lower the number of hash levels and the number of nodes in non-empty chains, and, in consequence, the lower the number of cache misses, on average. This explains why the FFP_{32} has better execution times than FFP_8 in Table 2 and the difference between our proposal and the best proposals on the search operation.

When the weight of removals is high, as in the 3rd benchmark, our FFP_{32} proposal is the third best proposal, behind the *CHM* and *NBHM* proposals. Again, this difference is explained by the number of hash levels that threads need to traverse to reach the level holding the node with the key being searched. For this particular benchmark, the FFP_{32} has a minimum and a maximum depth of 3 and 4, respectively.

Regarding scalability, in general, one can observe that all designs seem to have scalability problems (the best speedup of all experiments is just 14.10 obtained for the *CSL* design with 32 threads on the 4th benchmark). Additionally, the lowest base execution time is often associated with the lowest speedup ratios (for example, on the 2nd benchmark, where threads execute read-only operations, *CHM* clearly achieves the lowest base execution time, but the speedup ratios are consistently very low). Recall that the environment for our experiments was a SMP/NUMA based architecture. A SMP system is a *share everything* system where processors work under the supervision of a single operating system and where memory accesses use a common bus or inter-connect path. This means that, as we increase the number of threads in a 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, the efficiency of the intermediate levels of memory is lower and in some situations can even have a negative impact in the performance. These bottlenecks are analyzed and discussed in detail in Drepper's work [28].

Since our benchmark sets have irregular data requests, the probability of using intermediate levels of memory to satisfy data requests is not as high as expected. Analyzing again Table 2, in general, the *CSL* and *CT* proposals show the best speedup ratios. This mostly happens because, in general, they also show the worst base times. Our FFP_8 configuration consistently has better speedups than FFP_{32} which, again, can be explained by the higher base times of FFP_8, which suggests that our proposal is not able to maintain similar speedups for different configurations. This is explained by the fact that FFP_8 creates more hash levels than FFP_{32} and the time required to traversing them is higher in FFP_8 than in FFP_{32}. Anyhow, both FFP_8 and FFP_{32} configurations showed quite competitive speedups, clearly in line with the remaining proposals.

In summary, the results on Table 2 show that our proposal is quite competitive, when compared against other state-of-the-art proposals and, in particular, whenever the weight of the insert operation is high compared to the search and remove operations, our proposal shows the best execution times. For mixed insert, search and remove operations, our proposal stays in line with the remaining proposals but, if considering only the other lock-free approaches, *NBHM* and *CT*, then our FFP_{32} configuration showed the best execution times in almost all benchmarks and thread configurations.

7. Conclusions & Further Work

We have presented a novel, simple and scalable hash map design that fully supports the concurrent search, insert and remove operations. To the best of our knowledge, this is the first concurrent hash map design that puts together being lock-free and using fixed size data structures with persistent memory references, which we consider to be characteristics that have the best trade-off between performance, correctness and computational environment independence. Our design can be easily implemented in any type of language, library or within other complex data structures.

Experimental results show that our proposal is quite competitive when compared against other state-of-the-art proposals implemented in Java. Its design is modular enough to allow different types of configurations aimed for different performances in memory usage and execution time.

In future work, we plan to implement our proposal as an external library in order to be easily included in bigger systems, such as the Yap Prolog system [29], where the characteristics of being lock-free and using fixed size data structures with persistent memory references are key restrictions for the efficiency of the system. Further work also includes studying a *lock-free compression* operation that would compress clusters of internal hash levels, thereby reducing the average depth of the trie data structure, and an *elastic hashing* scheme that would remove empty hash levels, such that, the average depth when searching for keys which are not present in the hash map would be significantly reduced.

Acknowledgments

This work was funded 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.

References

[1] P. Bagwell, Ideal Hash Trees, Es Grands Champs 1195.

[2] E. Fredkin, Trie Memory, Communications of the ACM 3 (1962) 490–499.

[3] M. Areias, R. Rocha, An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs, in: International Conference on Parallel and Distributed Systems, IEEE Computer Society, 2012, pp. 636–643.

[4] P. Tsigas, Y. Zhang, Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors, SIGMETRICS Perform. Eval. Rev. 29 (1) (2001) 320–321.

[5] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism, " O'Reilly Media, Inc.", 2007.

[6] H. Sundell, P. Tsigas, NOBLE: Non-blocking Programming Support via Lock-free Shared Abstract Data Types, SIGARCH Comput. Archit. News 36 (5) (2009) 80–87.

[7] The java concurrency package (JSR-166).

[8] M. Areias, R. Rocha, Towards a Lock-Free, Fixed Size and Persistent Hash Map Design, in: M. Valero, A. Melo (Eds.), Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2017), IEEE Computer Society, Campinas, Brazil, 2017, pp. 145–152.

[9] M. Areias, R. Rocha, On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs, in: International Symposium on Practical Aspects of Declarative Languages, no. 8324 in LNCS, Springer, 2014, pp. 168–183.

[10] M. Areias, R. Rocha, A lock-free hash trie design for concurrent tabled logic programs, International Journal of Parallel Programming 44 (3) (2016) 386–406.

[11] P. Moreno, M. Areias, R. Rocha, Memory Reclamation Methods for Lock-Free Hash Tries, in: R. Ferreira, E. Ayguade (Eds.), Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2019), IEEE Computer Society, Campo Grande, Brazil, 2019, pp. 188–195.

[12] P. Moreno, M. Areias, R. Rocha, A Compression-Based Design for Higher Throughput in a Lock-Free Hash Map, in: M. Malawski, K. Rzadca (Eds.), Proceedings of the 26th International European Conference on Parallel and Distributed Computing (Euro-Par 2020), LNCS, Springer International Publishing, Warsaw, Poland, 2020, pp. 458–473.

[13] C. Click, Towards a Scalable Non-Blocking Coding Style (2007).
URL http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf

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

[15] M. Herlihy, J. M. Wing, Axioms for Concurrent Objects, in: ACM Symposium on Principles of Programming Languages, ACM, 1987, pp. 13–26.

[16] M. Herlihy, N. Shavit, On the Nature of Progress, in: Principles of Distributed Systems, Vol. 7109 of LNCS, Springer, 2011, pp. 313–328.

[17] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: Double-ended queues as an example, in: International Conference on Distributed Computing Systems, ICDCS '03, IEEE Computer Society, Washington, DC, USA, 2003.

[18] S. Prakash, Y. Lee, T. Johnson, Non-Blocking Algorithms for Concurrent Data Structures, Tech. Rep. TR91-002, Department of Computer and Information Sciences, University of Florida (1991).

[19] G. Barnes, A method for implementing lock-free shared-data structures, in: ACM Symposium on Parallel Algorithms and Architectures, SPAA '93, ACM, 1993.

[20] M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Trans. Program. Lang. Syst. 15 (5).

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

[22] 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.

[23] O. Shalev, N. Shavit, Split-Ordered Lists: Lock-Free Extensible Hash Tables, Journal of the ACM 53 (3) (2006) 379–405.

[24] J. Triplett, P. E. McKenney, J. Walpole, Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming, in: USENIX Annual Technical Conference, USENIX Association, 2011, p. 11.

[25] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, Communications of the ACM 33 (6) (1990) 668—-676.

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

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

[28] U. Drepper, What Every Programmer Should Know About Memory - Version 1.0, Tech. rep., Red Hat, Inc. (2007).

[29] V. Santos Costa, R. Rocha, L. Damas, The YAP Prolog System, Journal of Theory and Practice of Logic Programming 12 (1 & 2) (2012) 5–34.