Towards a Lock-Free, Fixed Size and Persistent Hash Map Design

Miguel Areias and Ricardo Rocha
CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
Email: {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 concurrent hash map design 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. 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.**

I. 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 type/size are (pre-)allocated within a page [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. 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. 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 this work, we propose a novel lock-free, fixed size and persistent hash map design for shared memory architectures, aimed to be as competitive as the existent alternative designs. 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 [4], [5], we have already proposed different concurrent hash map designs but only for the search and insert operations. 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.

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 a set of experiments comparing our design against other state-of-the-art concurrent hash map proposals, namely, C. Click *Non Blocking Hash Maps* [6], Prokopec *et al. Concurrent Tries* [7], and the *Concurrent Hash Maps* and *Concurrent Skip Lists* from the Java concurrency package. At the end, we present conclusions and further work directions.

II. Background & Related Work

The CAS instruction is at the heart of many lock-free data structures [8]. 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. In the work [9], Herlihy and Shavit presented a *grand unified explanation* for the progress properties, using *linearizability* which is an important correctness condition for the implementation of concurrent data structures [10].

The first correct CAS-based lock-free list-based set proposal was introduced by Harris [11]. Later, Michael improved Harris work by presenting a proposal that was compatible with all lock-free memory management methods and Michael used this proposal has the building block for lock-free hash maps [12]. Shalev and Shavit extended Michael's work when they presented their lock-free algorithm for resizing hash maps [13]. The algorithm is based in 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. 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 [14]. Concurrent non-blocking skip lists were later implemented by Herlihy *et al.* [15].

Regarding concurrent hash trie data structures, recently Prokopec *et al.* presented the *CTries* [7], 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.

III. 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 $w * i$ least significant 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. 1 showing a small example that illustrates how the concurrent insertion of nodes is done in a hash level.

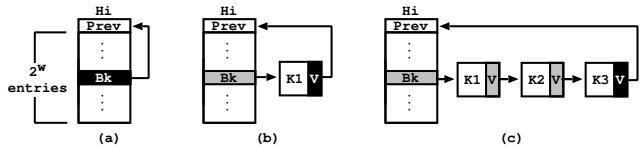

Fig. 1. Insert operation in a hash level

Figure 1(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. 1(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 1(b) shows the hash configuration after the insertion of node K_1 on the bucket entry B_k and Fig. 1(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. 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. In a worst case scenario, one would have MAX_NODES key collisions up to the last hash level (where the keys must be different, otherwise they would be the same). Assuming that keys have b bits then, in a worst case scenario, the hash trie maximum depth is $\frac{b}{w}$ and, for independent random n keys, it is expected to converge to a perfect hash trie map with a $\log n / \log 2^w$ depth. Starting from the configuration in Fig. 1(c), Fig. 2 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. 2(a)). The new hash level 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_{i+1} (Fig. 2(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

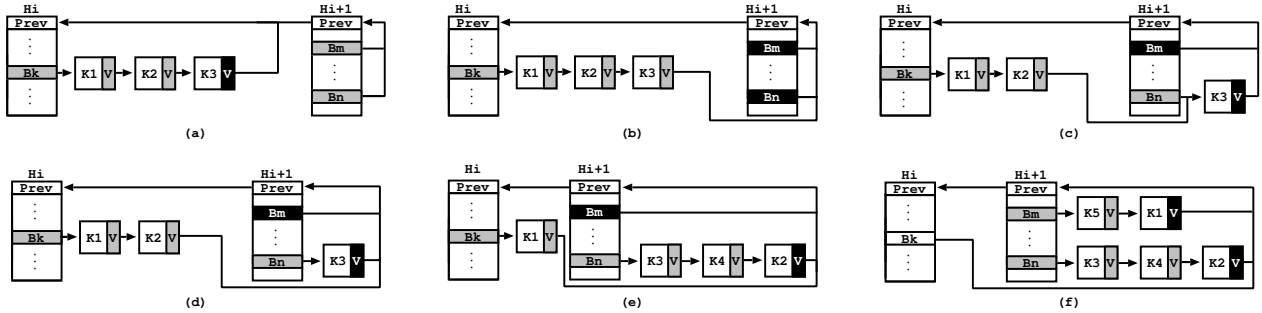

Fig. 2. Expand operation from a bucket entry to a second level hash with concurrent insertion of nodes

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 2(c) to 2(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 2(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. 2(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. 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_{i+1} (B_m in the figure) and then removed from the original chain, meaning in this case that the bucket entry B_k itself becomes a reference to the second level hash H_{i+1} (Fig. 2(f)). From now on, B_K is also 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. 2(e), where a node K_4 is inserted before K_2 in B_n and in Fig. 2(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. 1(c), where the initial condition of all keys is valid, Fig. 3 illustrates how the concurrent removal of nodes is done.

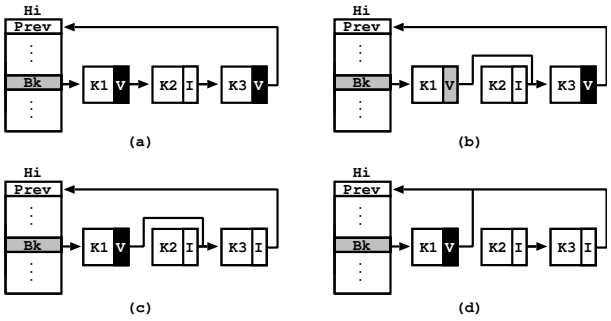

Fig. 3. Remove operation in a hash level

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. 3(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. 3(b), where node K_1 is chained to node K_3, thus bypassing node K_2 (invisibility step).

Figure 3(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. 3(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. 3(d) is the hash level H_i.

We conclude the presentation with Fig. 4 showing a last situation where a thread T is executing the expand operation (like in the example of Fig. 2) and another thread U is removing a key from the nodes being adjusted. Figure 4(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

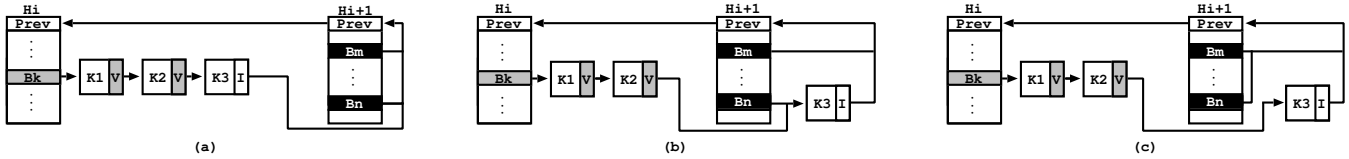

Fig. 4. Expand operation from a bucket entry to a second level hash with concurrent removal of nodes

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 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. 4(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. 4(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}.

IV. 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 remapping 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. 4 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. 3 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 16–22). For that, it uses three auxiliary variables: CVR holds the candidate valid reference where new insertions/adjustments should take place; $CVRN$ holds the chain reference of CVR; and C counts the number of valid nodes in the chain. Initially, CVR is the bucket entry from where the chain starts, R and $CVRN$ 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 (lines 48–50). 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 (lines 48–50), 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 CVR 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[1]. This is necessary since we need to guarantee that the node is valid when updating a chain reference.

Before trying to insert N after the CVR 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 CVR from $CVRN$ 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, *AdjustNodeOnHash()* 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 $newH$ and continues (line 39).

Next, we present the procedure that supports the remove operation. Algorithm 2 shows the pseudo-code for the

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

Algorithm 1 *AdjustNodeOnHash(node N, hash H)*

```
1: ForceCAS(NextRef(N), H)
2: if IsInvalidNode(N) then // invalid nodes are left unchanged
3:   return
4: B ← GetHashBucket(H, Hash(Level(H), Key(N)))
5: if EntryRef(B) = H then // B is an empty bucket
6:   if CAS(EntryRef(B), H, N) then
7:     if IsInvalidNode(N) then
8:       MakeNodeInvisible(N, H)
9:     return
10: R ← EntryRef(B)
11: if IsHash(R) then // R references a second hash level
12:   return AdjustNodeOnHash(N, R)
13: CVR ← B
14: CVRN ← R
15: C ← 0
16: repeat // traverse the chain of nodes
17:   if IsValidNode(R) then // update CVR if node is valid
18:     CVR ← R
19:     CVRN ← NextRef(CVR)
20:     C ← C + 1
21:   R ← NextRef(R)
22: until IsHash(R)
23: if R = H then // chain ended in the same hash level
24:   if C = 0 then // no valid chain nodes found
25:     if CAS(EntryRef(B), CVRN, N) then
26:       if IsInvalidNode(N) then
27:         MakeNodeInvisible(N, H)
28:       return
29:     else // another thread made progress in the meantime
30:       R ← EntryRef(B)
31:   else // a valid node was found
32:     if C = MAX_NODES then // chain is full
33:       newH ← AllocInitHash(Level(H) + 1)
34:       PrevHash(newH) ← H
35:       if CAS(Next(CVR), (CVRN, Valid), (newH, Valid)) then
36:         ExpandToNewHash(newH, N, H)
37:         return AdjustNodeOnHash(N, newH)
38:       else // another thread gained access to the expand operation
39:         FreeHash(newH)
40:     if CAS(Next(CVR), (CVRN, Valid), (N, Valid)) then
41:       if IsInvalidNode(N) then
42:         MakeNodeInvisible(N, H)
43:       return
44:     else // another thread made progress in the meantime
45:       R ← NextRef(CVR)
46:   if IsNode(R) then
47:     return AdjustNodeOnHash(N, H)
48: while Level(R) > Level(H) + 1 do // move to hash level after H
49:   R ← PrevHash(R)
50: return AdjustNodeOnHash(N, R)
```

Algorithm 2 *SearchRemoveKeyOnHash(key K, hash H)*

```
1: B ← GetHashBucket(H, Hash(Level(H), K))
2: if EntryRef(B) = H then // B is an empty bucket
3:   return
4: R ← EntryRef(B)
5: if IsHash(R) then // R references a second hash level
6:   return SearchRemoveKeyOnHash(K, R)
7: CVR ← B
8: CVRN ← R
9: repeat // traverse the chain of nodes
10:   if IsValidNode(R) then
11:     if Key(R) = K then // found key in R
12:       if MakeNodeInvalid(R) then
13:         return MakeNodeInvisible(R, H)
14:     else
15:       CVR ← R
16:       CVRN ← NextRef(CVR)
17:   R ← NextRef(R)
18: until IsHash(R)
19: if R = H then // chain ended in the same hash level
20:   if CVR = B then // no valid chain nodes found
21:     R ← EntryRef(B)
22:   else // a valid node was found
23:     R ← NextRef(CVR)
24:   if R = CVRN then // no progress in the meantime
25:     return
26:   if IsNode(R) then
27:     return SearchRemoveKeyOnHash(K, H)
28: while Level(R) > Level(H) + 1 do // move to hash level after H
29:   R ← PrevHash(R)
30: return SearchRemoveKeyOnHash(K, R)
```

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).

Otherwise, it starts traversing the chain of nodes searching for a valid node R with K (lines 10–17). 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 CVR and $CVRN$ 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, which means that no expansion operation is taking place at the same time, the algorithm needs to confirm that no other thread has changed the chain (lines 20–27). Otherwise, R refers a deeper hash level, case in which the algorithm is restarted in the hash level after H (lines 28–30). To confirm that nothing has changed, it updates R with the chain reference in CVR (lines 20–23) and compares it against $CVRN$ (line 24). If they are different, that means that, in the meantime, some change occurred in R, and the process is restarted in the same hash level H (line 27) or in the hash level after H (lines 28–30), case R references a deeper hash level.

Finally, Alg. 3 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

Algorithm 3 *MakeNodeInvisible(node N, hash H)*

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

structures BVR (*before valid reference*) and AVR (*after valid reference*), respectively before and after N in the chain of nodes, in order to bypass node N by chaining BVR to AVR.

The algorithm begins by setting R and AVR 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 R refers a deeper hash level, the process is restarted in the hash level after H (lines 23–25). Otherwise, the algorithm ended in the same hash level H (line 5) and it proceeds to compute the valid data structure BVR 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 $BVRN$ the chain reference of BVR (line 10).

At the end of the traversal, if R reaches N then we are in condition to bypass N by chaining BVR to AVR and thus make N invisible (lines 14–20). For that, the algorithm applies a CAS operation to BVR trying to update it from the reference saved in $BVRN$ to AVR and keeping the node condition as valid if BVR is a node (line 18). Notice that if the CAS operation fails, then it means that the BVR node has changed somewhere between the instant where it was found valid and the CAS execution. In such case, the process is restarted (line 20), 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, if R is H that means that N is already invisible, thus the algorithm simply returns (lines 21–22). Otherwise, R refers a deeper hash level and the process is restarted in the hash level after H (lines 23–25).

V. 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 machine with 32-Core AMD Opteron (TM) Processor 6274 (2 sockets with 16 cores each) with 32GB of main memory, each processor with caches L1, L2 and L3 respectively with sizes of 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)* [6] and Prokopec *et al. Concurrent Tries (CT)* [7][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 I 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 I
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[3] containing different benchmark sets of 10^6 randomized items, with each set divided in three operations: (i) insertion of new items; (ii) search of items; and (iii) removal of items. To spread threads among a 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 different operations on S. For the search and remove operations, the corresponding items are inserted beforehand and without counting to the execution time. To

[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

TABLE II

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

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 II shows the results 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 II 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 levels for the FFP_8 and FFP_{32} proposals and we noticed that, if we subtract such time to the overall execution time, we got execution times similar to those of *CHM*.

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 II 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, the CSL and CT proposals show the best speedup ratios. This mostly happens because they also show the worst base times, generally. Our FFP_8 configuration consistently has better speedups than FFP_{32} which, again, can be explained by the worst base times of FFP_8. Anyway, both FFP_8 and FFP_{32} configurations showed quite competitive speedups which are clearly in line with all the remaining proposals.

In summary, the results on Table II 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.

VI. 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 [16], 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.

Acknowledgments

Work funded by ERDF through Project 9471-RIDTI and the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT as part of project UID/EEA/50014/2013. Miguel Areias was funded by the FCT grant SFRH/BPD/108018/2015.

References

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

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

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

[5] ——, "A lock-free hash trie design for concurrent tabled logic programs," *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 386–406, 2016.

[6] C. Click, "Towards a Scalable Non-Blocking Coding Style," 2007. [Online]. Available: http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf

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

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

[9] M. Herlihy and N. Shavit, "On the Nature of Progress," in *Principles of Distributed Systems*, ser. LNCS. Springer, 2011, vol. 7109, pp. 313–328.

[10] M. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

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

[12] 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.

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

[14] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668—-676, 1990.

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

[16] V. Santos Costa, R. Rocha, and L. Damas, "The YAP Prolog System," *Journal of Theory and Practice of Logic Programming*, vol. 12, no. 1 & 2, pp. 5–34, 2012.