2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications

On Extending a Fixed Size, Persistent and Lock-Free Hash Map Design to Store Sorted Keys

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*—Searching is a crucial time-consuming part of many programs, and using a good search method instead of a bad one often leads to a substantial increase in performance. 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 concurrent hash map design that fully supports the concurrent search, insert and remove operations on hash tries designed to store sorted keys. To the best of our knowledge, our design is the first concurrent hash map design that puts together the following characteristics: (i) use fixed size data structures; (ii) use persistent memory references; (iii) be lock-free; and (iv) store sorted keys. Experimental results show that our design is quite competitive when compared against other state-of-the-art designs implemented in Java.**

***Index Terms*—Hash Tries, Lock-Freedom, Sorting.**

I. Introduction

Hash maps is a very common and efficient data structure used to store data that can be organized as pairs (K, C), where the mapping between the unique key K and the associated content C is given by a hash function. Hash tries (or hash array mapped tries) are a tree-based data structure with nearly ideal characteristics for the implementation of hash maps [1]. 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/compressing it in order to deal with hash collisions.

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.

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 hash map data structure. The traditional approach to synchronize access to critical sections in concurrent data structures is to use locking primitives, such as, spinlocks, mutexs or semaphores. Lock-free techniques offer several advantages over the traditional 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 [3] and they are available in several different frameworks, such as, Intel's Threading Building Blocks [4], the NOBLE library [5] or the Java concurrency package [6].

Traditional hash maps do no store sorted keys, which makes them unsuitable for non-exact match searches, such as to find all keys in an interval. Searching is a crucial time-consuming part of many programs, and using a good search method instead of a bad one often leads to a substantial increase in performance [7]. The earliest search algorithm — binary search — was first mentioned by John Mauchly [8] more than six decades ago, 25 years before the advent of relational databases [9]. Nowadays, one of the most critical database primitives is tree-structured index search, which is used for a wide range of applications where low latency and high throughput matter, such as data mining, financial analysis, scientific workloads and more [10].

In this work, we propose a novel concurrent hash map design that puts together the following characteristics: (i) use fixed size data structures; (ii) use persistent memory references; (iii) be lock-free; and (iv) store sorted keys. Our proposal is based on hash tries to implement fixed size data structures with persistent memory references, on single-word CAS (compare-and-swap) instructions to implement lock-freedom, and on *xor* operations to assist in sorting the hash values corresponding to keys. In previous work [11], we have already proposed a concurrent hash map design which supports most of the characteristics above with the exception of sorting. This work extends that previous design to also support sorting of keys. To materialize the new design, we had to redesign the existent search, insert and expand operations. To the best of our knowledge, none of the available alternatives in the literature fulfills all these four characteristics simultaneously.

The remainder of the paper is organized as follows. First, we introduce relevant background and present the key ideas of our design. Next, we discuss in more 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. At the end, we present conclusions and further work directions.

978-1-7281-1141-4/18/$31.00 ©2018 IEEE
DOI 10.1109/BDCloud.2018.00070

II. The Skeleton of Our Design

Our design combines hashing with sort and tree search algorithms. It is based on hash tries, which include *hash arrays of buckets* and *leaf nodes*. The leaf nodes store the key/content pairs and the hash arrays of buckets implement a hierarchy of hash levels of fixed size 2^W. To map a key K into this hierarchy, 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 $W * I$ highest 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 should be stored. For the sake of simplicity of presentation, we will consider the identity function for hashing (i.e., $H = hash(K) = K$) and we will only show the key part of the key/content pair in the figures that follow. Figure 1 shows the configuration of a hash trie for keys with 3 bits length (values from 0 to 7) and hash levels of size 2, i.e., with chunks of $W = 1$ bit.

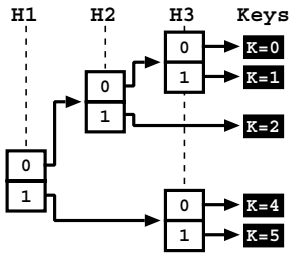

Fig. 1. Sorting keys with hash tries

As keys are inserted using their high-order bits for each level, they become immediately sorted (from top to down in Fig. 1) in the hash trie. The hash levels are only expanded when inserting a key in a full bucket entry. To expand a level H_i with a full bucket entry B, we apply a *xor* operation between the new key and the existent key in B to check in which chunk of bits the keys first differ. If they differ in a higher chunk of bits than the hash level i, then we insert a new hash level in a deeper level (we call this *front-expansion*). Otherwise, we insert a new hash level in a shallow level (we call this *back-expansion*). Figure 2 shows an example of both expansions as a result of inserting the keys 3 (front-expansion) and 6 (back-expansion) in the hash trie of Fig. 1.

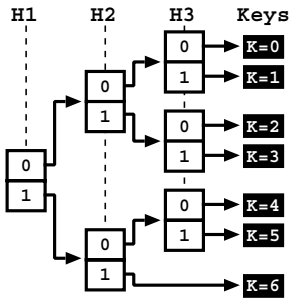

Fig. 2. Inserting keys 3 (front-expansion) and 6 (back-expansion)

The insertion of key 3 (011 in binary) collides with the bucket entry for key 2 (010 in binary) in the second hash level, which results in the xor operation leading to $011 \oplus 010 = 001$. The two keys first differ in the third bit chunk (from left to right), thus leading to a front-expansion (chunk $3 >$ level 2). Similarly, the insertion of key 6 (110 in binary) collides with the bucket entry for key 4 (100 in binary) in the third hash level, which results in the xor operation $110 \oplus 100 = 010$. The two keys first differ in the second bit chunk (chunk $2 <$ level 3), thus leading to a back-expansion.

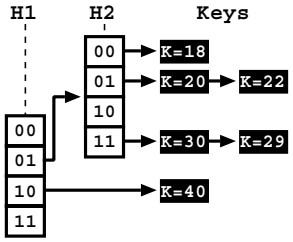

Fig. 3. Combining chunks of bits with chains of keys

The base configuration shown in Fig. 1 and Fig. 2, with chunks of $W = 1$ bit and one key per bucket entry (i.e., without hash collisions), can be easily adapted to different configurations. Figure 3 shows a new configuration of a hash trie for keys with 6 bits length (values from 0 to 64), hash levels of size 4, i.e., with chunks of $W = 2$ bits (00, 01, 10 and 11 in binary) and allowing collisions up to 4 keys. In this scenario, the hash levels are also expanded only when inserting a key in a full bucket entry, i.e., with a chain of 4 keys in this case. These chains can be important to amortize the number of hash levels per path. By default, the insertion of keys in a chain is done at the end of the chain. Thus, chains might not have sorted keys after insertion and a small sorting operation might be required to sort the keys in a chain, if the number of keys is higher than one.

In a worst case scenario, one would have 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 and have a $\mathcal{O}(\log_B N)$ complexity.

III. Our Design By Example

In this section, we focus the discussion on the key aspects of our design, namely on how concurrent operations work in a lock-free fashion. We begin with Fig. 4 showing a small example that illustrates how the concurrent insertion of nodes is done in a hash level.

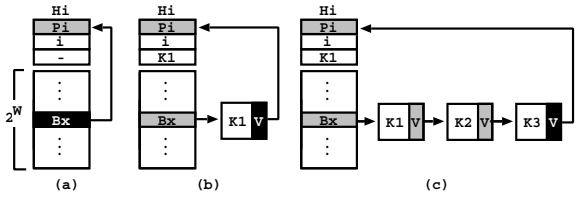

Fig. 4. Insert operation in a hash level

Fig. 4(a) shows the initial configuration for a hash level H_i. Each hash level is formed by a bucket array of 2^W entries (as mentioned before) and by a header, which includes a backward reference to the previous hash level, a hash level chunk identifier and a key representative of the hash level, respectively, values P_i, i and K_1 in Fig. 4 (in Fig. 4(a) the key representative is marked as '–' since the hash level is still empty). For the root level, the backward reference is $Null$.

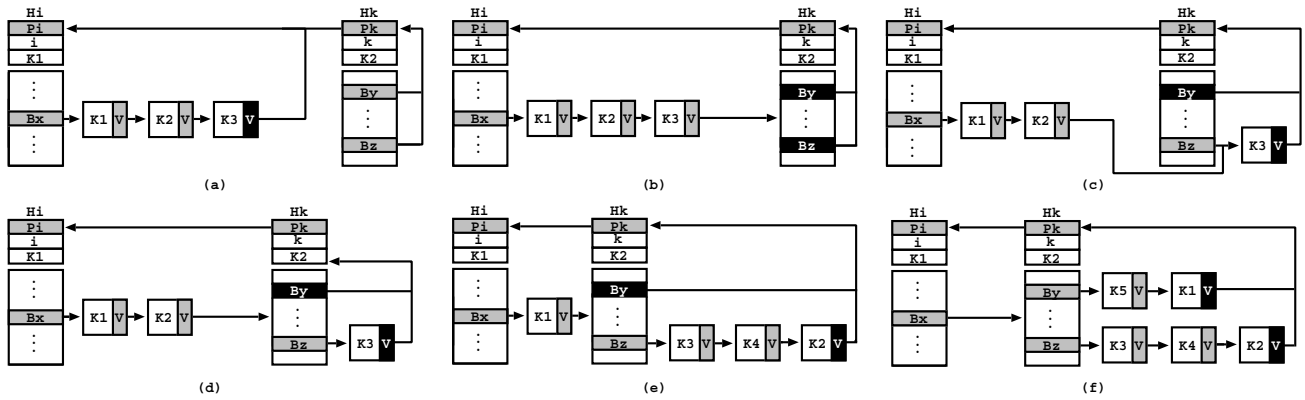

Fig. 5. Front-expansion with the concurrent insertion of nodes

The bucket entries are initialized with a reference to the current hash level. In Fig. 4(a), B_x represents a particular bucket entry of H_i. Each bucket entry stores either a reference to a hash level or a reference to a separate chaining mechanism, using a chain of leaf nodes, that deals with the hash collisions for that entry. Each leaf node includes a tuple that holds both a reference to a next-on-chain leaf node and the condition of the node, which can be valid (V) or invalid (I). The initial condition of a node is valid. Figure 4(b) shows the hash configuration after the insertion of node K_1 on the bucket entry B_x and Fig. 4(c) shows the hash configuration after the insertion of nodes K_2 and K_3 also in B_x. The insertion of nodes is done at the end of the chain and a new inserted node closes the chain by referencing back the current hash level.

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.

Starting from the configuration in Fig. 4(c), Fig. 5 illustrates the *front-expansion* operation to a second level hash for the bucket entry B_x. The front-expansion operation is activated whenever a thread T 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 corresponding to the number of collisions allowed (in what follows, we consider a threshold value of three keys). In such case, T starts by pre-allocating a second level hash H_k, with all entries referring the respective level and with a key representative consisting of the key in the chain (K_2 in the example of Fig. 5) which differs in the lower chunk of bits from the new key that is being inserted by T.

The new hash level H_k is then used to implement a synchronization point with the current IP (node K_3 in Fig. 5(a)) that will correspond to a successful CAS operation trying to update H_i to H_k (Fig. 5(b)). From this point on, the insertion of new nodes on B_x will be done starting from the new hash level H_k. If the CAS operation fails, that means that another thread has gained access to the IP and, in such case, T aborts its front-expansion operation. Otherwise, T starts the remapping process of placing the valid nodes K_1, K_2 and K_3 in the correct bucket entries in the new level. Figures 5(c) to 5(f) show the remapping sequence in detail. For simplicity of illustration, we will consider only the entries B_y and B_z on level H_k and assume that K_1, K_2 and K_3 will be remapped to these bucket entries. In order to ensure lock-free synchronization, 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.

Fig. 5(c) shows the hash trie configuration after the successful CAS operation that adjusted node K_3 to entry B_z. After this step, B_z passes to the gray state and K_3 becomes the next IP for the insertion of new nodes on B_z. Note that the initial chain for B_x has not been affected yet, since K_2 still refers to K_3. Next, on Fig. 5(d), the chain is adjusted and K_2 is updated to refer to the second level hash H_k. The process then repeats for K_2 (the new last node on the chain for B_x). First, K_2 is remapped to entry B_z and then it is removed from the original chain, meaning that the previous node K_1 is updated to refer to H_k (Fig. 5(e)). Finally, the same idea applies to node K_1. In the continuation, K_1 is also remapped to a bucket entry on H_k (B_y in the figure) and then removed from the original chain, meaning in this case that the bucket entry B_x itself becomes a reference to the second level hash H_k (Fig. 5(f)). Concurrently with the remapping process, other threads can be inserting nodes in the same bucket entries for the new level. This is shown in Fig. 5(e), where a node K_4 is inserted before K_2 in B_z and in Fig. 5(f), where a node K_5 is inserted before K_1 in B_y.

We describe next the *back-expansion* operation. The back-expansion operation has a lower priority than the front-

expansion operation, i.e., a back-expansion only begins if no front-expansion is undergoing. If a front-expansion is undergoing and a thread T wants to execute a back-expansion, T begins by assisting the threads doing the front-expansion and only then it begins the back-expansion. Figure 6 shows a possible sequence of steps involving a back-expansion operation concurrently with a front-expansion operation.

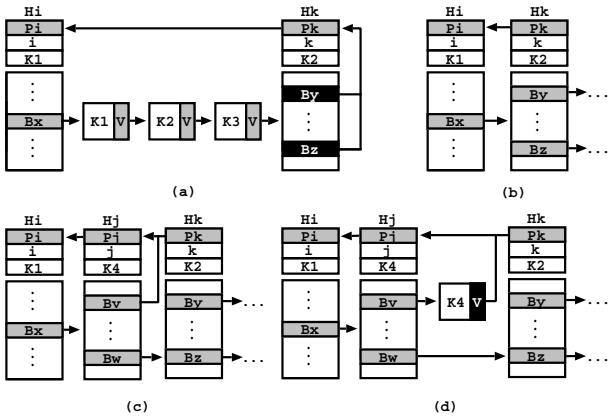

Fig. 6. Back-expansion concurrently with a front-expansion

Figure 6(a) shows the initial configuration, where a front-expansion operation is undergoing for the bucket entry B_x to a second level hash H_k, and Fig. 6(b) shows the end of the front-expansion, after the CAS operation updating B_x to H_k. Assume now that during the front-expansion, a new thread T reaches B_x looking to insert a key K_4 and that $K4 \oplus K2$ (note that K_2 is the key representative of H_k) leads to a bit chunk level j lower than k. Thus, a third hash level H_j must be included between H_i and H_k.

Figure 6(c) shows the resulting configuration after the insertion of the hash level H_j. The initialization of a hash level in a back-expansion is sightly different from the front-expansion, since one of the bucket entries must refer the front hash level H_k. T uses the key representative of the front hash level, K_2 in Fig. 6(c), to compute the bucket entry B_w that should refer to H_k. All the remaining bucket entries are initialized referring back the respective level H_j. At the end of the initialization step, T applies a CAS operation on B_x setting it to refer to H_j. Finally, T can insert the key K_4 in the hash level H_j. Figure 6(d) shows this final configuration.

It is important to notice that while H_j is being initialized, another thread can insert first a different hash level H_l in B_x. In such case, the CAS operation on B_x will fail and T must recover, considering now that the front hash level is H_l instead of H_k. Then, T either moves to H_l if $j >= l$ or, otherwise, retries to insert H_j, now between H_i and H_l, after updating the bucket entries as described above.

IV. Algorithms

In this section, we present the key algorithms that implement our design. We begin with Algorithms 1 and 2 showing the pseudo-code for the search/insert operation of a given key K in a given hash level H. Both algorithms execute recursively, moving through the hierarchy of hash levels, until K is found or inserted in H (for the entry call, H is the root level). Algorithm 1 deals with the hash level data structures and Algorithm 2 deals with the chain of leaf nodes.

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

```
1:  B ← GetBucket(H, Hash(K), ChunkId(H))
2:  R ← EntryRef(B)
3:  if IsHash(R) then  // R references a hash level
4:    if ChunkId(R) <= ChunkDiffer(K ⊕ KeyRep(R)) then
5:      return SearchInsertOnHash(K, R)
6:    else
7:      return BackExpansion(K, H, R)
8:  else  // R references a leaf node
9:    T ← ⟨B, R, 0, MAXCHUNK⟩
10:   return SearchInsertOnChain(K, H, T)
```

In more detail, Algorithm 1 starts by applying the hash function to K that allows obtaining the appropriate bucket entry B of H that fits with the corresponding chunk of bits (line 1). Next, the algorithm reads the reference R on the head of B (line 2) and checks whether it references a hash level or a leaf node. If R references a hash level (lines 4–7), then the algorithm uses K and the key representative of R to find: (i) if R can hold K, case in which it calls itself using now R for the hash level (line 5), or (ii) if a back-expansion operation must be executed (line 7). The back-expansion operation is presented later on Algorithm 4. Otherwise, R references a leaf node, in which case it calls Algorithm 2 to deal with the chain of leaf nodes (lines 9–10).

Algorithm 2 receives an extra argument T, in the form of a quadruplet, corresponding to information regarding the leaf node where the traversal should start. Starting from T, the algorithm initializes a set of auxiliary variables (line 1): CVR holds the candidate valid reference where new insertions should take place; $CVRN$ holds the chain reference of CVR; C counts the number of valid nodes in the chain; and CHK holds the lowest chunk of bits found. When called from Algorithm 1 at line 10, CVR is the bucket entry from where the chain starts, $CVRN$ is the first node on the chain, C is 0 and CHK is $MAXCHUNK$.

After this initialization step, the algorithm traverses the chain of leaf nodes until it finds a reference to a hash level (lines 3–13). While traversing, the algorithm only considers valid nodes. If the key K is found in a node, the algorithm returns the corresponding node R (lines 5–6). Otherwise, the auxiliary variables are updated accordingly (lines 7–11).

At the end of the traversal, the algorithm checks if R ended in the same hash level H, which means that no expansion operation is taking place at the same time, and it proceeds trying to insert K (lines 15–41). Otherwise, R refers a deeper hash level, case in which a front-expansion is going on, thus it moves to assist in it (line 43). The front-expansion operation is presented later on Algorithm 3.

If R ended in the same hash level then two situations might occur: no valid chain nodes were found (lines 16–21) or,

Algorithm 2 *SearchInsertOnChain(Key K, Hash H, Tuple T)*

```
1: ⟨CVR, CVRN, C, CHK⟩ ← T
2: R ← CVRN
3: repeat // traverse the chain of leaf nodes
4:   if IsValidNode(R) then
5:     if Key(R) = K then
6:       return R
7:     CVR ← R
8:     CVRN ← NextRef(CVR)
9:     C ← C + 1
10:    if CHK > ChunkDiffer(K ⊕ KeyRep(R)) then
11:      CHK ← ChunkDiffer(K ⊕ KeyRep(R))
12:  R ← NextRef(R)
13: until IsHash(R)
14: if R = H then // chain ended in the same hash level
15:   if C = 0 then // no valid chain nodes found (empty bucket)
16:     newN ← AllocInitNode(K, H, Valid)
17:     if CAS(EntryRef(CVR), CVRN, newN) then
18:       return newN
19:     else
20:       FreeNode(newN)
21:       return SearchInsertOnHash(K, H)
22:   else // at least one valid node was found
23:     if C = MAXNODES then // chain is full
24:       newH ← AllocInitHash(H, CHK, K)
25:       if CAS(Next(CVR), (CVRN, valid), (newH, valid)) then
26:         return FrontExpansion(K, H)
27:       else
28:         FreeHash(newH)
29:         return FrontExpansion(K, H)
30:     else // C ≠ MAXNODES
31:       newN ← AllocInitNode(K, H, Valid)
32:       if CAS(Next(CVR), (CVRN, valid), (newN, valid)) then
33:         return newN
34:       else
35:         FreeNode(newN)
36:         R ← NextRef(CVR)
37:         if IsHash(R) then
38:           return SearchInsertOnHash(K, H)
39:         else
40:           T ← ⟨CVR, R, C, CHK⟩
41:           return SearchInsertOnChain(K, H, T)
42: else // R ≠ H
43:   return FrontExpansion(K, H)
```

at least, a valid node was found (lines 23–41). If no valid nodes were found, then a new leaf node $newN$ representing K is allocated and properly initialized (line 16). Then, the algorithm tries to insert K on the head of CVR by using a CAS operation that updates it to $newN$ (line 17). If the CAS succeeds, $newN$ was successfully inserted and the algorithm ends by returning it (line 18). Otherwise, in case of CAS failure, the head of CVR has changed in the meantime, so the bucket entry is not empty anymore, and it calls back the *SearchInsertOnHash()* algorithm (line 21).

Otherwise, at least one valid node was found, thus the algorithm checks: (i) if the chain is full, or (ii) if the chain can support another node. If the chain is full, then a new hash level $newH$ is allocated and properly initialized (line 24). Next, the algorithm applies a CAS operation on CVR, trying to update its next reference to $newH$ (line 25), and executes the front-expansion operation (note that if the CAS fails, that means that a front-expansion is already going on, case in which it still moves to assist in it). Otherwise, if the chain is not full, a new leaf node $newN$ is created to hold the key K, and the algorithm tries to insert $newN$ in the head of the chain (line 32). If the CAS succeeds, it returns $newN$ (line 33). Otherwise, in case of CAS failure, the algorithm reads the next reference R in CVR (line 36). If R is a hash level, the process is restarted in the same hash level H (line 38). Otherwise, R is a leaf node, and the process is restarted in the same chain starting now from R (line 41).

Algorithm 3 shows the process of front-expansion. The algorithm starts by obtaining the bucket entry B from where the chain starts and by reading the reference R on the head of B (lines 1–2). Next, it follows the chain of leaf nodes until it finds a second hash level R (lines 3–4). If the second level hash R is not immediately after H, that means that the front-expansion operation has been already completed by others. Otherwise, if the second level hash R is immediately after H, it calls the *AdjustNodesToSecondLevelHash()* procedure (line 6), which implements the remapping process of placing the valid nodes on H in the correct bucket entries of R (as illustrated in Fig. 5). We will not show the details about this procedure, since it is quite similar to the *SearchInsertOnChain()* procedure. In both scenarios, at the end, the algorithm calls again the *SearchInsertOnHash()* procedure, thus restarting the process of inserting the key K on H.

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

```
1: B ← GetBucket(H, Hash(K), ChunkId(H))
2: R ← EntryRef(B)
3: while IsNode(R) do
4:   R ← NextRef(R)
5: if PrevHash(R) = H then
6:   AdjustNodesToSecondLevelHash(H, R)
7: return SearchInsertOnHash(K, H)
```

Finally, Algorithm 4 presents the process of back-expansion (for a better understanding, please remember the example shown in Fig. 6). Note that the third argument NH represents the hash level immediately after H, as called from Algorithm 1 at line 7.

Again, the algorithm starts by obtaining the bucket entry B from where the chain starts and by computing the chunk difference between K and the key representative of NH in order to allocate and initialized a new hash level $newH$ (lines 1–3). The bucket entry of $newH$ corresponding to the key representative of NH is then made to refer to NH (lines 4–5). Next, the algorithm applies a CAS on the bucket entry B trying to update NH to $newH$ (line 6). If the CAS succeeds, then the algorithm tries to update the previous field from NH to refer to the new hash level $newH$ using a second CAS operation and, in the continuation, it calls the

Algorithm 4 *BackExpansion(Key K, Hash H, Hash NH)*

1: $B \leftarrow GetBucket(H, Hash(K), ChunkId(H))$
2: $CHK \leftarrow ChunkDiffer(K \oplus KeyRep(NH))$
3: $newH \leftarrow AllocInitHash(H, CHK, K)$
4: $newB \leftarrow GetBucket(newH, Hash(KeyRep(NH)), CHK)$
5: $EntryRef(newB) \leftarrow NH$
6: **if CAS**$(EntryRef(B), NH, newH)$ **then**
7: **CAS**$(PrevHash(NH), H, newH)$
8: **return** $SearchInsertOnHash(K, newH)$
9: **else**
10: $FreeHash(newH)$
11: **return** $SearchInsertOnHash(K, H)$

SearchInsertOnHash() procedure in order to restart the process of inserting the key K but now on the new hash level $newH$ (lines 7–8). Note that if this second CAS operation (at line 7) fails, that means that another back-expansion operation was executed in the meantime, leading to the insertion of another hash level between $newH$ and NH. In such case, the update of the previous field of NH is of the responsibility of this back-expansion in between. Otherwise, if the CAS at line 6 fails, that means that another back-expansion succeeded in the meantime and in such case the process is restarted by calling again the *SearchInsertOnHash()* algorithm with K and H.

V. Performance Analysis

This section presents experimental results comparing our design with other state-of-the-art concurrent hash map designs. The environment for our experiments was a SMP (Symmetric Multi-Processing) system, based in a NUMA (Non-Uniform Memory Access) architecture 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 jdk-10.0.1.

Although our design is platform independent, we have chosen to make its first implementation in Java, mainly for two reasons: (i) rely on Java's garbage collector to reclaim 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 design against Prokopec *et al. CTries (CT)*[1], a non-blocking concurrent hash trie design based on shared-memory single-word CAS instructions [12], [13]. 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.

We have ran our design with a threshold value of 3 chain nodes for the hash collisions, 16 buckets entries per hash level, and with two configurations, the original design (as presented in [11]) and the new design with sorted keys. In what follows, we will name such designs as *Free Fixed Persistent Hash Map (FFP)* and those two configurations as FFP_O and FFP_S, respectively. To put all the designs in perspective, Table I shows how they support/implement the features of: (i) use fixed size data structures; (ii) maintain the access to all internal data structures as persistent memory references; (iii) be lock-free; and (iv) store sorted keys.

TABLE I
Features supported by the designs evaluated

Features / Designs	CHM	CSL	CT	FFP_O	FFP_S
Fixed size structures	✗	-	✓	✓	✓
Persistent references	✗	✓	✗	✓	✓
Lock-freedom	✗	✗	✓	✓	✓
Sorted Keys	✗	✓	✗	✗	✓

A. Experimental Results

For the experiments, we developed a testing environment[2] containing different benchmark sets of $3 * 10^6$ randomized items, with each set divided in four different operations: (i) insertion of new items; (ii) removal of items; (iii) search for existing items (i.e., to be found); and (vi) search for missing items (i.e., to be not found). To spread threads among a set S items, 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 remove and search for existing items 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 II shows the results obtained for the *CHM*, *CSL*, *CT*, FFP_O and FFP_S designs using six benchmark sets that vary in the percentage of concurrent operations to be executed. The 1st benchmark only performs inserts, the 2nd only removes, and the 3rd only searches for existing items. The remaining benchmarks perform mixed operations with different percentages of inserts, removes and searches. 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 design has it own advantages and disadvantages, i.e., there is no single design that overcomes all the remaining designs. For the execution times, the table shows a clear trade-off balance between the concurrent insertion, removal and search of items. The designs with the best execution times in the concurrent insertions are not so good in the concurrent removals and the same happens with the concurrent searches of items.

When the weight of insertions is high, as in the 1st and 5th benchmarks, the FFP_S design shows the best base times, i.e., with one thread. However, as we increase the number of

[1]Downloaded on January 18, 2016 from https://github.com/romix/java-concurrent-hash-trie-map/tree/master/src/main/java/com/romix/scala/collection/concurrent

[2]Available from https://github.com/miar/ffps

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, REMOVE AND SEARCH 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**	**CT**	**FFP**$_O$	**FFP**$_S$	**CHM**	**CSL**	**CT**	**FFP**$_O$	**FFP**$_S$
1st – Insert: 100%	**Remove: 0%**		**Search (existing items): 0%**			**Search (missing items): 0%**				
1	1,166	2,079	3,285	1,304	**1,019**					
8	771	560	745	**398**	697	1.51	3.71	**4.41**	3.28	1.46
16	729	348	573	**313**	608	1.60	**5.97**	5.73	4.17	1.68
24	913	**298**	588	366	623	1.28	**6.98**	5.59	3.56	1.64
32	869	**276**	531	317	765	1.34	**7.53**	6.19	4.11	1.33
2nd – Insert: 0%	**Remove: 100%**		**Search (existing items): 0%**			**Search (missing items): 0%**				
1	**385**	2,983	4,178	2,174	1,067					
8	**105**	905	607	470	633	3.67	3.30	**6.88**	4.63	1.69
16	**104**	525	452	350	294	3.70	5.68	**9.24**	6.21	3.63
24	**102**	447	436	424	428	3.77	6.67	**9.58**	5.13	2.49
32	**101**	455	334	343	191	3.81	6.56	**12.51**	6.34	5.59
3rd – Insert: 0%	**Remove: 0%**		**Search (existing items): 100%**			**Search (missing items): 0%**				
1	**198**	2,715	2,043	977	327					
8	**79**	451	359	196	151	2.51	**6.02**	5.69	4.98	2.17
16	**82**	319	228	163	171	2.41	8.51	**8.96**	5.99	1.91
24	**94**	325	196	172	174	2.11	8.35	**10.42**	5.68	1.88
32	**90**	409	230	162	170	2.20	6.64	**8.88**	6.03	1.92
4th – Insert: 0%	**Remove: 0%**		**Search (existing items): 50%**			**Search (missing items): 50%**				
1	**135**	1,874	1,258	815	288					
8	**55**	301	241	142	102	2.45	**6.23**	5.22	5.74	2.82
16	**59**	201	180	107	96	2.29	**9.32**	6.99	7.62	3.00
24	**66**	202	142	113	110	2.05	**9.28**	8.86	7.21	2.62
32	**70**	252	161	103	89	1.93	7.44	7.81	**7.91**	3.24
5th – Insert: 50%	**Remove: 0%**		**Search (existing items): 25%**			**Search (missing items): 25%**				
1	832	3,717	2,736	1,259	**786**					
8	688	539	493	**272**	396	1.21	**6.90**	5.55	4.63	1.98
16	475	341	301	**238**	351	1.75	**10.90**	9.09	5.29	2.24
24	519	295	261	**222**	390	1.60	**12.60**	10.48	5.67	2.02
32	395	307	236	**135**	573	2.11	**12.11**	11.59	9.33	1.37
6th – Insert: 20%	**Remove: 10%**		**Search (existing items): 35%**			**Search (missing items): 35%**				
1	505	3,709	2,457	996	**497**					
8	**183**	566	396	206	270	2.76	**6.55**	6.20	4.83	1.84
16	**88**	334	250	145	310	5.74	**11.10**	9.83	6.87	1.60
24	**106**	283	247	185	271	4.76	**13.11**	9.95	5.38	1.83
32	**96**	298	244	146	187	5.26	**12.45**	10.07	6.82	2.66

threads, it is not able to scale properly, showing a scalability ratio in line with the *CHM* design. The other three designs, *CSL*, *CT* and *FFP*$_O$, start from worst base times but are able to scale better as we increase the number of threads. Comparing with *FFP*$_O$, *FFP*$_S$ achieves better results for one thread because *FFP*$_O$ uses a hash function that consumes chunks of 4 bits ($2^4 = 16$ buckets entries per level) from the lowest to the highest bits, while the *FFP*$_S$ design has a hash function that disperses better the keys among the hash levels, once it uses a *xor* operation to detect where the keys differ from the highest to the lowest bits, which can potentially lead to an average small number of hash levels (trie depth). On the other hand, for the remaining thread launches, i.e., with more than one thread, *FFP*$_S$ is more suitable to cache-misses in a concurrent environment. This is because the *FFP*$_S$ design implements a hash header that has more fields than the one used in *FFP*$_O$ requiring reading the chunk identifier and the hash representative when traversing a hash level, while in the *FFP*$_O$ design these fields do not exist. Also, the previous field can be updated concurrently in *FFP*$_S$, while in the *FFP*$_O$ design it remains unchangeable.

On the other hand, when the weight of removals is high, as in the 2nd benchmark, or when the weight of search operations is high, as in the 3rd and 4th benchmarks, the *FFP*$_S$ design achieves the second best results, being only overcome by the *CHM* design. In the 3rd benchmark, the difference between *FFP*$_S$ and *CHM* remains stable, while in the 2nd and 4th benchmarks the difference tends to decrease, as we increase the number of threads. In these benchmarks, most of the time is spent traversing the hash trie, and this is where the *FFP*$_S$ design shows its advantages compared to the other designs and, in particular, compared to *CSL*, the other design also supporting sorted keys. Comparing with *FFP*$_O$, the *FFP*$_S$ design shows better results, meaning that it seems quite effective in avoiding traversing so many hash levels.

Regarding scalability, in general, the CSL and CT designs show the best speedup ratios. This mostly happens because they also show the worst base times, generally. The *FFP*$_O$ design consistently has better speedups than *FFP*$_S$, which again can be partially explained by the worst base times of *FFP*$_O$. Compared to *CHM*, *FFP*$_S$ shows similar speedups.

B. Scalability Issues

One can observe that all designs seem to have scalability problems, once the best speedup of all experiments is only 13.11 (obtained for the *CSL* design with 24 threads on the 5th benchmark). Additionally, the lowest base execution time is often associated with the lowest speedup ratio. For example, on the 3rd benchmark, where threads execute read-only operations, the speedups of the *CHM* and FFP_S designs are consistently very low.

To better understand these results, one must remember that the environment for our experiments was a SMP/NUMA based architecture. A SMP system is a *share everything* system where multiple processors are working under the supervision of a single operating system and all processors access memory using a common bus or inter-connect path. This means that, as we increase the number of processors in the computation, the bus becomes overloaded which can result in a performance bottleneck. NUMA tries to mitigate the burden of the main bus by adding intermediate levels of memory shared among some of the processors so that several data accesses do not need to travel on the main bus. However, on applications that have irregular data requests, the efficiency of the intermediate levels of memory is lower and in some situations can even have a negative impact in the performance. These SMP/NUMA bottlenecks are analyzed in detail in Drepper's work [14], where Drepper describes how data flows between processors, bus and memory controllers and shows benchmarks that present a clear performance degradation. For example, Drepper presents a read-only benchmark that has a performance degradation of 34% on the number of cycles required to satisfy data requests when using four threads against using one thread only.

Unfortunately, since applications using hash designs have irregular data requests, the probability of using intermediate levels of memory to satisfy data requests is not as high as expected. Thus, the higher the number of levels in a hash design the higher the probability of cache-misses. As an additional experiment for the search operation, we measured the time that threads spent just in hash trie levels for the FFP_S design and we noticed that, if we subtract such time to the overall execution time, we got execution times similar to those of *CHM*. As further work, we plan to extend our design with a *compress-on-removal* operation that compresses empty hash levels, thus reducing the potential number of data requests.

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, using fixed size data structures and persistent memory references with storing sorted keys, 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 design is quite competitive when compared against other state-of-the-art designs implemented in Java, with particular emphasis on removal and search operations, whenever most of the time is spent traversing the hash trie levels.

As further work, we plan to: (i) extend the API of our design to support the return of sets of keys, such as keys between two threshold values; (ii) extend the design with an additional *compress-on-removal* operation aimed to compress empty hash levels; (iii) evaluate the performance of our design in different architectures; and (iv) implement our design as an external library in order to be easily included in bigger systems, such as the Yap Prolog system [15], where the characteristics of our design and, in particular, the possibility of storing sorted keys, is a very important feature 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] P. Tsigas and Y. Zhang, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 29, no. 1, pp. 320–321, 2001.
[4] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.
[5] H. Sundell and P. Tsigas, "NOBLE: Non-blocking Programming Support via Lock-free Shared Abstract Data Types," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 80–87, 2009.
[6] "The java concurrency package (JSR-166)."
[7] D. E. Knuth, *The Art of Computer Programming: Volume 3: Sorting and Searching (2nd Ed.)*. Addison-Wesley Longman, 1998.
[8] J. Mauchly, *Theory and Techniques for Design of Electronic Digital Computers*, 1946.
[9] E. F. Codd, "A relational model for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.
[10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Designing fast architecture-sensitive tree search on modern multicore/many-core processors," *ACM Trans. Database Syst.*, vol. 36, no. 4, pp. 22:1–22:34, 2011.
[11] M. Areias and R. Rocha, "Towards a Lock-Free, Fixed Size and Persistent Hash Map Design," in *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Applications and Technologies (SBAC-PAD 2017)*, M. Valero and A. Melo, Eds. Campinas, Brazil: IEEE Computer Society, October 2017, pp. 145–152.
[12] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent Tries with Efficient Non-Blocking Snapshots," in *ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 151–160.
[13] A. Prokopec, "Cache-tries: Concurrent lock-free hash tries with constant-time operations," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 137–151.
[14] U. Drepper, "What Every Programmer Should Know About Memory - Version 1.0," Red Hat, Inc., Tech. Rep., 2007.
[15] 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.