A Compression-Based Design for Higher Throughput in a Lock-Free Hash Map *

Pedro Moreno[0000−0003−3745−5845], Miguel Areias[0000−0003−1589−3174], and Ricardo Rocha[0000−0003−4502−8835]

CRACS & INESCTEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{pmoreno,miguel-areias,ricroc}@dcc.fc.up.pt

Abstract. Lock-free implementation techniques are known to improve the overall throughput of concurrent data structures. A hash map is an important data structure used to organize information that must be accessed frequently. A key role of a hash map is the ability to balance workloads by dynamically adjusting its internal data structures in order to provide the fastest possible access to the information. This work extends a previous lock-free hash map design to also support *lock-free compression*. The main goal is to significantly reduce the depth of the internal hash levels within the hash map, in order to minimize cache misses and increase the overall throughput. To materialize our design, we redesigned the existent search, insert, remove and expand operations in order to maintain the lock-freedom property of the whole design. Experimental results show that lock-free compression effectively improves the search operation and, in doing so, it outperforms the previous design, which was already quite competitive when compared against the concurrent hash map design supported by Intel.

Keywords: Hash Maps · Lock-Freedom · Concurrency · Performance

1 Introduction

Hash maps are a very common and efficient data structure used to map keys to values, where the mapping between the unique key K and the associated value V is given by a *hash function*. Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash maps [3]. An essential property of the trie data structure is that common prefixes are stored only once [6], which in the context of hash maps allows us to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions.

* This work is funded by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. Pedro Moreno and Miguel Areias are funded by the FCT grants SFRH/BD/143261/2019 and SFRH/BPD/108018/2015, respectively.

However, trie-based hash maps are prone to generate higher cache misses than traditional hash maps, thus they tend to perform worse as the depth of the trie increases. Fortunately, tries are widely used in different domains and literature shows a significant amount of effort in studying their properties and implementations [9] and, in particular, for cache-based architectures, in studying how to mitigate cache effects to achieve better performance [1]. Recently, Li *et al.* studied the throughput of several kinds of hash map designs and presented a high-throughput and memory-efficient concurrent cuckoo-based hashing technique that supports multiple readers and writers [10].

Lock-freedom is an important concurrency technique that is known to improve the overall throughput of concurrent data structures. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. In particular, lock-free trie-based hash maps offer a viable alternative to memory-efficient hash-mapping [14, 2]. However, the cache misses problem was also observed by Prokopec *et al.* when they compared the CTries data structure [13], a lock-free trie-based hash map, against other state-of-the-art hash map designs.

Arguably, a well-know workaround to improve the performance of a trie-based data structure is to apply some sort of *compression technique* [11, 7] as a way to reduce the average depth of the trie data structure. Compression can be done at shallow or deeper trie levels, but a key advantage is that it can be done concurrently with the other operations. Two good examples are: (i) the B*-tree proposal [16], which supports a compression procedure that runs concurrently with regular operations, such as searches, insertions and removals, to merge nodes that are underfull; and (ii) the relaxed B-slack trees proposal [4] that supports a similar concurrent absorb operation that reduces the number of levels in the data structure.

In this work, we focus on extending a sophisticated implementation of a lock-free trie-based hash map, named *Lock-Free Hash Map (LFHT)* [12], to support *lock-free compression*. The original LFHT implements a hierarchy of hash levels whose branching factor is given by a fixed (and pre-defined) number of bucket entries per hash level. Traversing the hash levels in the LFHT data structure is $\mathcal{O}(\log_B K)$, where B represents the fixed number of bucket entries in a hash level and K is the overall number of keys inserted in the hash map. Our compress operation will be working on adjusting B to significantly reduce the average depth of the internal hash levels within the hash map, i.e., instead of a fixed number of bucket entries per hash level, we now support hash levels of different sizes. Compression is done incrementally, affecting well-defined clusters of hash levels, in order to meet varying (local) workloads. Since the number of levels to be traversed is expected to be lower, this reduces cache misses and increases the overall throughput. Experimental results show that lock-free compression effectively improves the search operation and, in doing so, it outperforms the previous design [12], which was already quite competitive, when compared against the concurrent hash map design in Intel's TBB library [15]. To materialize our design, we redesigned the existent search, insert, remove and expand operations in order to maintain the lock-freedom property of the whole design.

The remainder of the paper is organized as follows. First, we introduce some background regarding the LFHT design. Next, we discuss the main aspects of our design by example. Then, we describe implementation details and present the key algorithms required to easily reproduce our implementation by others. Finally, we show experimental results and end by outlining conclusions and further work.

2 Lock-Free Hash Tries

The LFHT data structure has two kinds of nodes: *hash nodes* and *leaf nodes*. The leaf nodes store key/value pairs and the hash nodes implement a hierarchy of hash levels of fixed size 2^w. To map a key/value pair *(k,v)* into this hierarchy, we compute the hash value h for k and then use chunks of w bits from h to index the appropriate hash node, i.e., for each hash level H_i, we use the i^{th} group of w bits of h to index the entry in the appropriate bucket array of H_i. To deal with collisions, the leaf nodes form a linked list in the respective bucket entry until a threshold is met and, in such case, an expansion operation updates the nodes in the linked list to a new hash level H_{i+1}, i.e., instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w. Figure 1 shows how the insertion of nodes is done in a hash level.

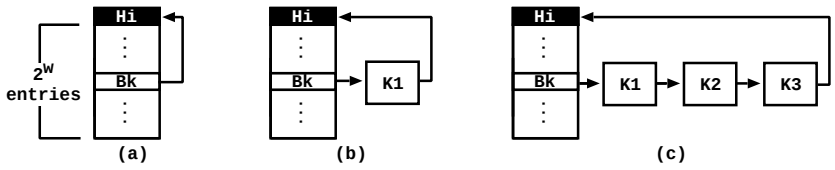


Fig. 1. Insertion of nodes in a hash level

Figure 1(a) shows the initial configuration of a hash level H_i. A hash level is formed by: (i) a hash node, which includes a header where control information is stored and a bucket array of 2^w entries; and by (ii) the corresponding chain of leaf nodes per bucket entry. Initially, all bucket entries are empty. In Fig. 1, B_k represents a particular bucket entry of H_i. A bucket entry stores either a reference to a hash node (initially the current hash node) or a reference to a separate chain of leaf nodes, corresponding to the hash collisions for that entry. Figure 1(b) shows the configuration after the insertion of node K_1 on B_k and Fig. 1(c) shows the configuration after the insertion of nodes K_2 and K_3. 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. A leaf node holds both a reference to a next-on-chain node and a flag with the condition of the node, which can be *valid* (V) or *invalid* (I). The initial condition of a node is valid and turns invalid when the node is marked for removal.

When the number of valid nodes in a chain reaches a given threshold, the next insertion causes the corresponding bucket entry to be expanded to a new hash

level (in what follows, we consider a threshold value of three). Figure 2 shows how nodes are remapped in the new level. The expansion operation starts by inserting a new hash node H_{i+1} at the end of the chain with all its bucket entries referencing H_{i+1} (as shown in Fig. 2(a)). From this point on, new insertions will be done on the new level H_{i+1} and the chain of leaf nodes on B_k will be moved, one at a time, to H_{i+1}. Figure 2(b) and Fig. 2(c) show how node K_3 is first remapped in H_{i+1} (bucket B_n) and then moved from H_i (bucket B_k). When the last node is moved, the bucket entry B_k in H_i is made to refer to the new hash node H_{i+1} (Fig. 2(d)).

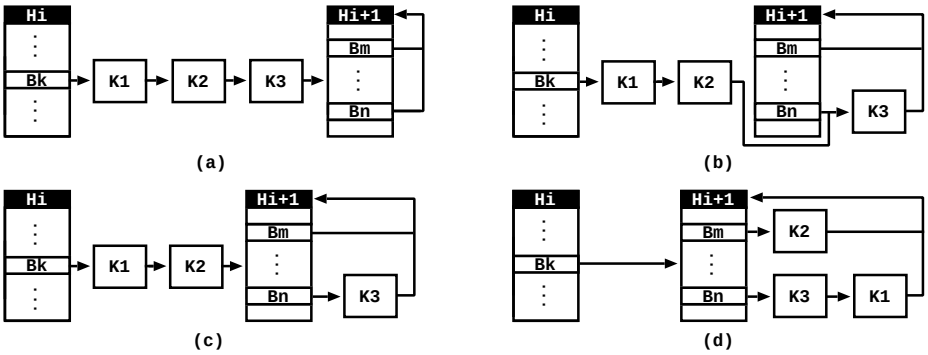


Fig. 2. Expansion of nodes in a hash level

In what follows, we base our work on the LFHT implementation [12] which supports the search, insert, remove and expand operations concurrently in a lock-free fashion and where threads collaborate to finish the undergoing expansions in a path before inserting new nodes. This implementation also supports a memory reclamation design, named *HHL (Hazard Hash and Level)*, that uses hazard pairs to define well-defined regions of memory to be protected from reclamation, which explores the characteristics of the LFHT data structure in order to achieve efficient memory reclamation with low and well-defined memory bounds.

3 Our Design by Example

In this section, we present our design by example. Our design takes advantage of the fine-grained and fully synchronized atomic CAS operation, which is at the heart of many lock-free data structures [8].

In a nutshell, the key idea of our design is to apply lock-free compression to clusters of hash nodes in order to reduce the average depth of hash levels needed to be traversed within the hash map. To activate compression on a cluster, the condition is to have a hash node (called the *head node* of the cluster of hash nodes) with all its bucket entries referring other hash nodes. A second condition is that the head node does not belong to a second cluster where another

compression is undergoing. If two or more compressions intersect, then priority is given to the compression whose head node has the lowest depth (i.e., near to the root of the hash map). Non-priority compressions are postponed (or aborted) until the top priority one completes. At the end of a compression, the cluster of hash nodes is replaced by a single hash node representing the cluster and the depth of any path traversing the cluster is reduced in one level.

Figure 3 shows an example of applying *lock-free compression* to a cluster of hash levels. For the sake of simplicity of illustration, we consider that hash nodes are initially allocated with two bucket entries and that R_1 to R_6 represent references to arbitrary hash or leaf nodes. Figure 3(a) shows the initial configuration where one can observe the existence of two clusters of hash nodes: cluster C_1 with head node H_i and including H_k and H_l; and cluster C_2 with head node H_k and including H_m and H_n. Since H_k, the head node of C_2, also belongs to C_1, priority is given to the compression of cluster C_1. Figure 3(b) shows the configuration after the compression of cluster C_1 where one can observe that H_i, H_k and H_l were replaced by a single new hash node H_x that has twice the size of bucket entries (four bucket entries in this case).

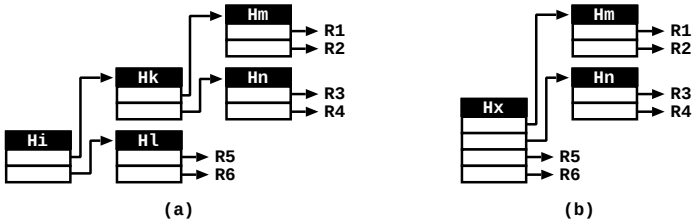


Fig. 3. Compression of a cluster of hash levels

Consider a thread traversing the configuration in Fig. 3 looking for reference R_3. Without compression (Fig. 3(a)), the thread begins by visiting H_i, then follows the reference in the first bucket to access H_k, next the reference in the second bucket to access H_n, and finally the reference in the first bucket to reach R_3. In the worst case, if the header and the corresponding bucket entry for each hash node do not fit inside the same cache line, reaching R_3 will require six memory accesses (two times the number of hash levels). After compression (Fig. 3(b)), the thread begins by visiting H_x and reaching R_3 requires one less hash level, corresponding to four memory accesses, in the worst case.

Let us consider now that lock-free compression is first triggered and successfully applied to C_2 and only then H_l is concurrently added to the hash map data structure to form cluster C_1. Figure 4(a) shows the resulting configuration, where one can observe that H_k, H_m and H_n were replaced by a single new hash node H_z with four bucket entries. As before, the access to references R_1, R_2, R_3 and R_4 were all reduced by one level, but the access to R_5 and R_6 remains unchanged and still requires traversing two hash levels. This illustrates one of

the advantages of prioritizing the compressions near the root of the hash map. A second advantage is that the application of compressions following the priority order of being near the root of the hash map converges to a *canonical structure*, while any other order of application can lead to different configurations at the end. A key motivation of lock-free compression is that regardless of which cluster is compressed first, the hash hierarchy will converge to a canonical structure. We next discuss how this is done in our design.

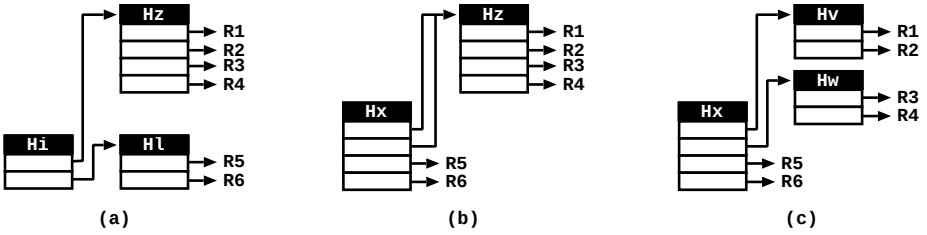


Fig. 4. Splitting of previously compressed hash levels

Starting from the configuration in Fig. 4(a), we now have a cluster C_1 formed by the head node H_i and including H_z and H_l. The problem is that, due to the fact that H_z already represents two hash levels as a result of a previous compression, H_z and H_l have a different number of bucket entries (4 and 2 entries, respectively) and, therefore, we cannot replace cluster C_1 by a single new hash node, as done previously. Figure 4(b) and Fig. 4(c) show two alternative approaches for compressing C_1 in this case.

The approach illustrated in Fig. 4(b) tries to preserve previous compressions. A new hash node H_x is introduced to represent C_1 (thus replacing H_i and H_l) but H_z is maintained. As intended, this approach succeeds in reducing the access to R_5 and R_6 in one level. However, since H_z represents two hash levels, the first two bucket entries of H_x are made to refer to H_z. This violates an invariant of the LFHT design, which requires not having more than one bucket entry referencing the same hash node, and makes it impossible to swap references to hash nodes with just a single word CAS operation.

The approach illustrated in Fig. 4(c) tries to preserve the canonical structure. Since H_z represents a less priority compression, it proceeds by undoing the previous compression and, for that, it splits H_z in two hash levels (H_v and H_w in Fig. 4(c)), each with half the bucket entries. Then, a new hash node H_x is still introduced to represent C_1, thus replacing H_i, H_l and part of H_z. As before, this approach succeeds in reducing the access to all references in one level, but now each bucket entry in H_x holds a reference to different hash nodes. One can observe that this configuration is similar to the one presented in Fig. 3(b), which represents the canonical form. This example shows that, regardless of the order of cluster compression, the hash hierarchy will converge to a canonical structure although, as in this situation, the compress operation would require extra steps.

4 Implementation Details

Starting from the high-level description of the previous section, we now discuss in more detail how lock-free compression is implemented on top of the LFHT data structure. Such detail is important since we want to show that lock-free compression is implemented by following a well-defined sequence of CAS operations. To implement lock-free compression, the following extensions were made to the LFHT data structure: (i) bucket entries now include a *freeze flag* that, when set, indicates that further updates cannot be made to the corresponding bucket entry; and (ii) the header of the hash nodes now includes a *compression representative* field, which refers to the new hash node representing the cluster being compressed, and a *compression count* field, which counts the number of bucket entries referring to hash nodes (and is used to trigger compression).

Figure 5 details the sequence of steps involved in the compression of a *standard* cluster of hash nodes, i.e., without splitting. For that, it considers a bucket entry B_k referring to a cluster with head node H_i and including H_k and H_l. As before, R_1, R_2, R_3 and R_4 represent references to arbitrary hash or leaf nodes.

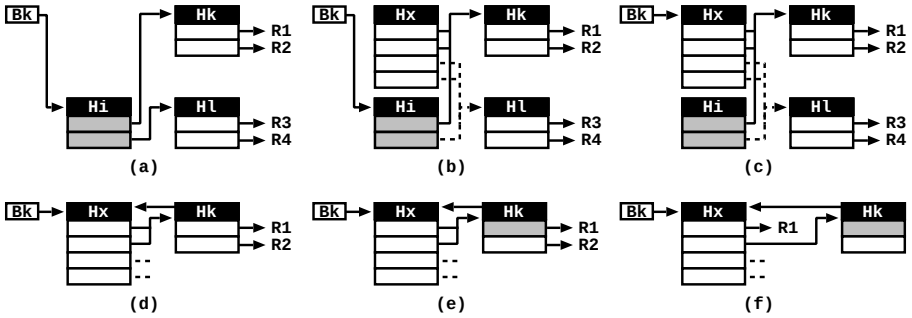


Fig. 5. A step by step compression operation without splitting

Figure 5(a) shows the first step of the compression procedure, where CAS operations are used to set the freeze flag of each bucket entry in the head node H_i (in what follows, frozen entries are marked gray). Remember that a frozen entry remains unchanged for the remaining lifetime. This freezing process is important because it implements the strategy where priority is given to the compression whose head node has the lowest depth. For example, if a less priority compression is being done on cluster with head node H_k, it will be aborted because it cannot update the corresponding first (frozen) bucket entry of H_i.

Next, Fig. 5(b) shows the second step of the compression procedure, where a new hash node H_x is first allocated and then initialized by copying the references from the bucket entries in H_i. In this case, since H_k and H_l are default sized (non-compressed) hash nodes, the size of H_x corresponds to doubling the size of H_i, and each pair of bucket entries in H_x is initialized to match the corresponding

H_i's entry. For example, the first two bucket entries of H_x are set to H_k, which is the reference in H_i's first entry, whereas the second two bucket entries of H_x are set to H_l, which is the reference in H_i's second entry. After this initialization, H_x is ready to be inserted in the LFHT data structure and, for that, a CAS operation is applied to B_k trying to replace H_i with H_x. Figure 5(c) shows the resulting configuration. It is important to notice that lock-freedom requires that, at any moment of the compression procedure, no thread can be blocked from traversing and accessing the available hash and leaf nodes. Figure 5(c) show us that, even in a scenario where a thread T is preempted in H_i, T is still able to traverse forward to the deeper levels H_k and H_l.

At this point, it is also important to notice that the configuration in Fig. 5(c) violates the invariant of not having more than one bucket entry referencing the same hash node. However, here, this is not a problem because the bucket entries in H_x are not yet the synchronization points for further updates on the cluster, since they are still referring to H_k and H_l. Thus, the next steps involve copying R_1 to R_4 from the bucket entries of H_k and H_l to the bucket entries of H_x. Figure 5(d) to Fig. 5(f) show how this is done for reference R_1. The same process applies to the remaining references (not shown here to simplify the illustration).

The next step is to set the new compression representative (header) fields of H_k and H_l to refer to H_x. Figure 5(d) shows the configuration after setting the compression representative field of H_k. The same process applies to H_l (not shown here to simplify the illustration). Note that copying the references R_1 and R_2 to H_x, will turn H_k invalid. The compression representative field implements a kind of reconnection path for invalid hash nodes. For example, in a scenario where a thread T is preempted in H_k and H_k turns invalid, the compression representative field allows T to recover to H_x.

The final steps involve freezing the first bucket entry of H_k, meaning that no further updates can be done there, and applying a CAS to the corresponding bucket entry in H_x in order to update it to R_1. Figure 5(e) shows the configuration after the freezing and Fig. 5(f) shows the configuration after the updating of R_1 in H_x. The same process is applied afterwards to the remaining bucket entries in H_x, adjusting R_2, R_3 and R_4, to finish the compression procedure. Note that these final steps do not violate the lock-freedom property of a search, insert, remove or expand operation being done concurrently, since the synchronization point in H_k is being moved to the corresponding bucket entry in H_x. In other words, an operation that would require updating the frozen bucket entry in H_k, will now follow the compression representative field to reach H_x and change the corresponding bucket entry there.

We conclude this section by describing a second compression situation, but now for a scenario leading to the splitting of previously compressed hash levels, as illustrated in Fig. 4. Figure 6 details the sequence of steps involved in the compression of a cluster with head node H_i and including H_z and H_l, where H_z is already the result of a previous compression.

As before, Fig. 6(a) shows the first step of the compression procedure, where CAS operations are used to set the freeze flag of each bucket entry in the head

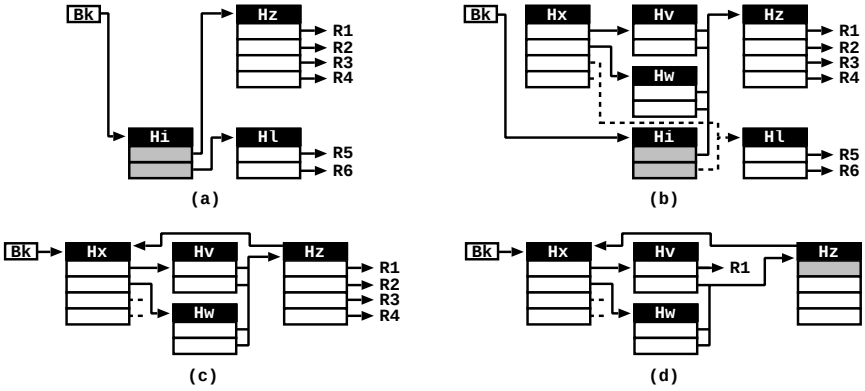


Fig. 6. A step by step compression operation with splitting

node H_i. Then, Fig. 6(b) shows the second step of the compression procedure, where new hash nodes H_x, H_v and H_w are first allocated (H_v and H_w representing the splitting of H_z in two hash levels, each with half the bucket entries) and then initialized by copying the references from the bucket entries in H_i. Next, Fig. 6(c) shows the configuration after updating the compression representative field of H_z to refer to H_x. The same process applies to H_l (not shown here to simplify the illustration). Note that H_v and H_w are not set as representative as, in general, this would require not a single representative field but an array of representatives (equal to the number of bucket entries per hash node). Finally, Fig. 6(d) shows the configuration after freezing the first bucket entry of H_z and after applying a CAS to the corresponding bucket entry in H_v in order to update it to R_1. The same process is then applied to the remaining bucket entries in H_v and H_w, adjusting references R_2 to R_6, to finish the compression procedure.

5 Algorithms

This section presents the key algorithms required to easily reproduce our implementation[1]. We begin with Alg. 1 to show the pseudo-code for the lock-free compression procedure for a given head node H_i.

Algorithm 1 *Compression*(*hash node* H_i)

```
1: FreezeBucketEntries(H_i)
2: H_x ← CompressionInit(H_i)
3: if CompressionCommit(H_i, H_x) then
4:     CompressionReps(H_x)
5:     CompressionRefs(H_x)
```

[1] Available from https://gitlab.com/pedromoreno/lfht-hhl/

FreezeBucketEntries() starts by implementing the first step of the compression procedure, as shown in Fig. 5(a) and Fig. 6(a). Then, *CompressionInit*() implements the second step, as shown in Fig. 5(b) and Fig. 6(b). Next, the conditional call to *CompressionCommit*() implements the step where H_i is replaced by H_x, as shown in Fig. 5(c). If it fails, meaning that there is an overlapping high priority compression being done, then H_x is simply deallocated. Otherwise, *CompressionReps*() sets the compression representative fields, as shown in Fig. 5(d) and Fig. 6(c), and *CompressionRefs*() updates the references in the bucket entries of the new hash nodes, as shown in Fig. 5(e–f) and Fig. 6(d). Pseudo-code for *CompressionInit*(), *CompressionReps*() and *CompressionRefs*() is presented in more detail in Alg. 2, 3 and 4, respectively.

Algorithm 2 *CompressionInit*(*hash node* H_i)

```
1: H_x ← AllocHashNode(HashSize(H_i) × HS)
2: for i ← 0 to HashSize(H_i) do
3:     H_j ← H_i.bucket[i]
4:     if HashSize(H_j) = HS then
5:         for j ← 0 to HS do
6:             H_x.bucket[i × HS + j] ← H_j
7:     else {splitting case}
8:         for j ← 0 to HS do
9:             H_v ← AllocHashNode(HashSize(H_j) ÷ HS)
10:            H_x.bucket[i × HS + j] ← H_v
11:            for v ← 0 to HashSize(H_v) do
12:                H_v.bucket[v] ← H_j
13: return H_x
```

Algorithm 3 *CompressionReps*(*hash node* H_x)

```
1: i ← 0
2: while i < HashSize(H_x) do
3:     H_k ← H_x.bucket[i]
4:     if HashLevel(H_k) ≠ HashLevel(H_x) then {splitting case}
5:         H_k ← H_k.bucket[0]
6:     H_k.compr_representative ← H_x
7:     i ← i + HS
```

In these algorithms, *HS* is the default number of bucket entries for a standard hash node, *HashSize*() returns the number of bucket entries in a hash node, and *HashLevel*() returns the initial depth of a hash node. In Alg. 4, the *compr_count* field counts the number of bucket entries in a hash node referring to deeper hash nodes and is used to trigger lock-free compression when all bucket entries are referring to deeper hash nodes (lines 18–21 in Alg. 4).

Algorithm 4 *CompressionRefs(hash node H_x)*

```
1: xCount ← 0
2: for x ← 0 to HashSize(H_x) do
3:     H_k ← H_x.bucket[x]
4:     if GetLevel(H_k) = GetLevel(H_x) then
5:         R ← FreezeBucketEntry(&(H_k.bucket[x mod HS]))
6:         CAS(&(H_x.bucket[x]), H_k, R)
7:         if IsHash(R) then
8:             xCount ← xCount + 1
9:     else
10:        xCount ← xCount + 1
11:        kCount ← 0
12:        for k ← 0 to HashSize(H_k) do
13:            H_z ← H_k.bucket[k]
14:            R ← FreezeBucketEntry(&(H_z.bucket[(x mod HS) × HashSize(H_k) + k]))
15:            CAS(&(H_k.bucket[k]), H_z, R)
16:            if IsHash(R) then
17:                kCount ← kCount + 1
18:        if AtomicAdd(H_k.compr_count, kCount = HashSize(H_k) then
19:            Compression(H_k)
20: if AtomicAdd(H_x.compr_count, xCount) = HashSize(H_x) then
21:     Compression(H_x)
```

6 Performance Analysis

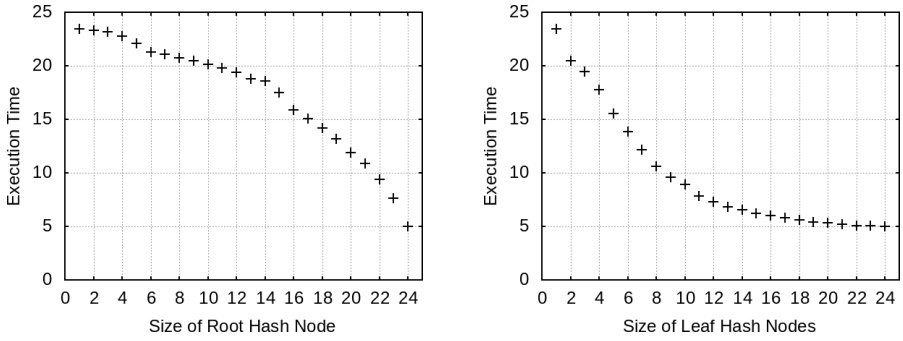
The environment for our experiments was a SMP system based in a NUMA architecture with two Intel Xeon X5650, each having 6 cores (12 hyperthreads) at 2.66GHz, 12MB Intel Smart Cache, 96GB of main memory, and running the Linux kernel 4.15.0-72. To measure execution time, all programs were compiled with GCC 9.2.0 with -O3 and using the jemalloc memory allocator 5.0 [5]. We ran each benchmark 5 times and took the mean of those runs.

6.1 Compression Benefits

Compression benefits heavily rely on the memory environment where we are running our benchmarks. Factors like cache sizes, placement policies, prefetching optimizations can have a significant impact on the overall performance of the LFHT design. To put our results in perspective, first we ran a specific benchmark designed to address the potential gains that one would expect to have when using compression. For that, we used a static version of the LFHT design that implements fixed predefined configurations of hash levels, with a different number of bucket entries on each hash node, and we measured the execution time for one thread performing only search operations on those configurations.

Starting from a maximal configuration of 24 uncompressed hash levels, all with the same minimal size of 2^1 bucket entries, we studied the effect of applying two different types of compression operations: (i) by reducing the number of hash levels from the root hash node to the leaf hash nodes; and (ii) by reducing the number of hash levels from the leafs to the root. Figure 7 shows the

execution time, in seconds, for executing 2^{24} search operations with one thread when reducing the number of hash levels in both directions (Fig. 7(a) for the root to leafs compression and Fig 7(b) for the leafs to root compression) until reaching the configuration with just a single hash node with 2^{24} bucket entries. The x-axis represents the number of hash levels compressed in a configuration. In both figures, the x-axis value of 1 represents the maximal configuration of 24 uncompressed hash levels and the x-axis value of 24 represents the single fully compressed hash node with 2^{24} bucket entries. The other x-axis values represent intermediate configurations. For example, the x-axis value of 10, in Fig. 7(a) represents the configuration whose first hash node includes 2^{10} bucket entries followed by 14 uncompressed hash levels, and in Fig. 7(b) represents the configuration with 14 initial uncompressed hash levels followed by a final hash node with 2^{10} bucket entries.



(a) Compression from root to leafs (b) Compression from leafs to root

Fig. 7. LFHT's compression effects for 2^{24} search operations with one thread

In Fig. 7(a), one can observe that, for root hash nodes with less than 2^{14} bucket entries, the benefits are small, but then, for higher compression ratios, the results show a significant impact on reducing the execution time. This happens because most of the execution time is spent on waiting for swaps between the different levels of memory and because the hash nodes closest to the root tend to remain in cache. Consequently, compressing the first 14 levels only reduces the amount of cache accesses, which results in a poor impact on the total executing time. On the other hand, further compression is able to reduce effectively the number of memory accesses and memory swaps.

In Fig. 7(b), one can observe that compressions up to a size of about 2^{10} are quite effective in reducing the execution time, whereas after that size they are not as much. This can be explained by the fact that, after a certain size, the benefits of compression are absorbed by the caching effects.

As a result of this study, in what follows, we have chosen to set the root hash node of the LFHT design with 2^{16} bucket entries, thus ensuring that compressions would have an impact in the execution time. This will create a memory

overhead, which can be considered negligible, since it amounts to just 512KB. All the other hash nodes, allocated during execution, begin with 2^4 bucket entries, which is the minimum size allowed by the original LFHT design.

6.2 Performance Results

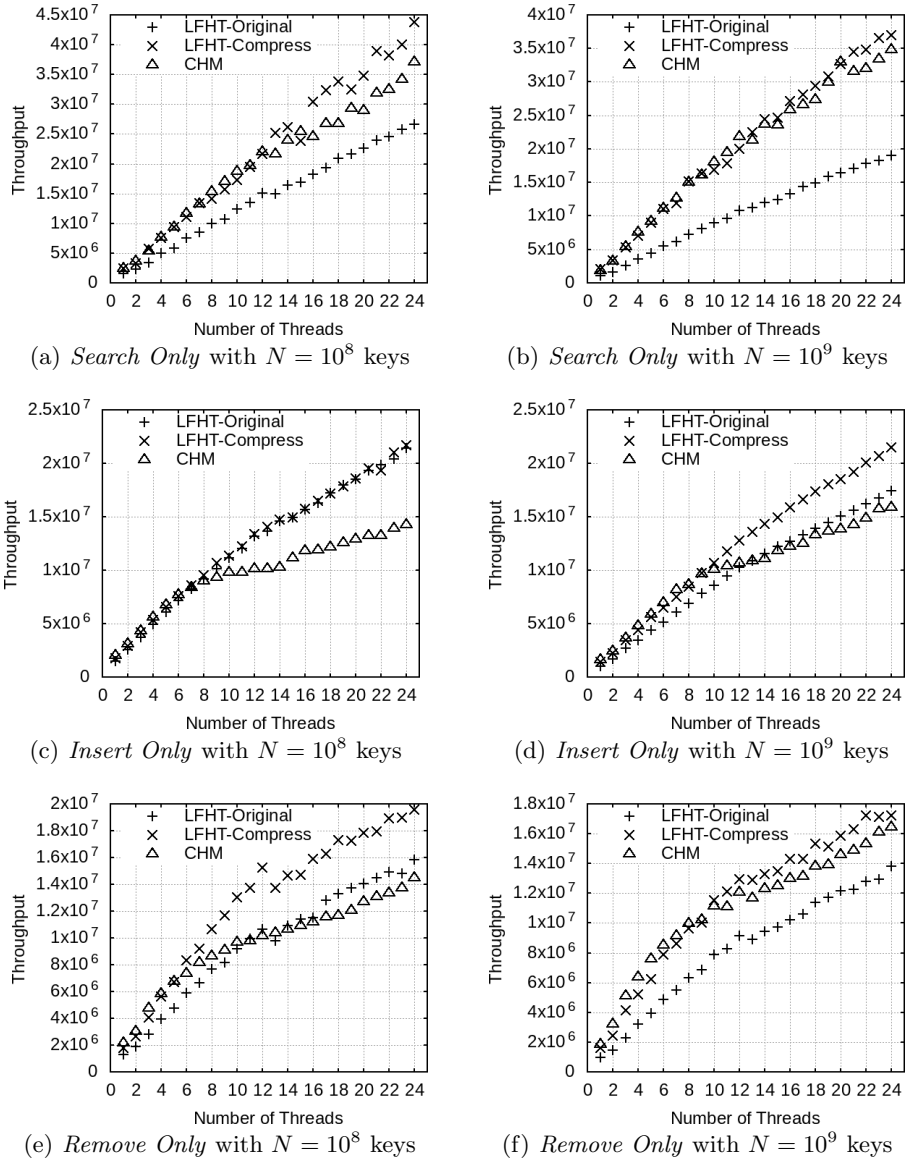
In this subsection, we analyze the performance of our compression design in three different scenarios: (i) *Search Only*, where threads search for N keys in a hash map with the N keys inserted; (ii) *Insert Only*, where threads insert N keys in an empty hash map; and (iii) *Remove Only*, where threads remove N keys in a hash map with N random keys.[2] On each scenario, we used two sets of N random keys, namely 10^8 and 10^9 keys. To support concurrent randomicity on each thread, we used glibc PRNG (Pseudo Random Number Generator), such that, for insertions we just insert random keys by giving each thread a different seed, and for search and remove, we reuse the seeds used for insertion, ensuring that we search or remove each key only once. Although these scenarios are not real-world applications, they do provide a strong insight about the expected behavior of the design. Note that, since hash-maps use hash functions to disperse keys among the internal data structures, we argue that real-world applications should provide similar results to the ones that we present next.

Figure 8 shows throughput results (higher is better) comparing our compressed design (LFHT-Compress) against the original design (LFHT-Original), and the Concurrent Hash Map design (CHM) of Intel-TBB library [15], when running a number of threads from 1 to 24 with 10^8 and 10^9 keys in the three previously mentioned scenarios.

Figure 8(a) and Fig. 8(b) show throughput results for the *Search Only* scenario. Comparing the two LFHT designs, one can observe that LFHT-Compress obtains improvements against LFHT-Original of around 50% with 10^8 keys and around 100% with 10^9 keys. When comparing against CHM, LFHT-Compress has almost always the best results, with CHM very close. This can be explained by the fact that the final configuration of both designs is quite similar, since CHM also uses only a root hash level to do the initial scatter of keys.

Figure 8(c) and Fig. 8(d) show throughput results for the *Insert Only* scenario. Comparing the two LFHT designs, one can observe that both achieve similar results for 10^8 keys but LFHT-Compress is clearly better for 10^9 keys. Even though LFHT-Compress is doing more work by compressing hash levels, it is able to improve the overall throughput. This happens because the cost of doing extra work on compression is compensated by the shorter paths leading to the insertion points. When comparing with CHM, LFHT-Compress has almost always the best results, however in this scenario the difference is more significant as we increase the number of threads. One reason that can explain this

[2] We have also tested other scenarios that mix the search, remove and insert operations, but have not obtained relevant results. This can be explained by the fact that the interference between different types of operations is rare enough to not impact performance.



(a) *Search Only* with $N = 10^8$ keys

(b) *Search Only* with $N = 10^9$ keys

(c) *Insert Only* with $N = 10^8$ keys

(d) *Insert Only* with $N = 10^9$ keys

(e) *Remove Only* with $N = 10^8$ keys

(f) *Remove Only* with $N = 10^9$ keys

Fig. 8. Throughput for the *Search Only*, *Insert Only* and *Remove Only* scenarios

difference is the fact that, since CHM is lock-based, it seems unable to scatter the concurrency spots as we increase the number of threads, since each lock is being used to block a large portion of paths within the hash map. On the other hand, since LFHT-Compress is lock-free, it is able to control the concurrency spots with the fine grain given by the CAS operation.

Finally, Fig. 8(e) and Fig. 8(f) show throughput results for the *Remove Only* scenario. Comparing the two LFHT designs, one can observe that LFHT-Compress is again better than LFHT-Original and that the difference increases as we increase the number of threads. This can be explained by the gains observed for LFHT-Compress on the search operation. When comparing with CHM, LFHT-Compress is again better by far than CHM in the 10^8 scenario, with the difference increasing as the number of threads increases, whereas in the 10^9 scenario the difference is almost constant. This can be explained by the same reasons mentioned before for the *Insert Only* scenario (lock-based vs lock-free).

7 Conclusions & Further Work

We have presented a novel lock-free compression design for a lock-free trie-based hash map, named LFHT, that is able to significantly reduce the depth of the internal hash levels within the hash map structure. By doing so, our design is able to minimize cache misses and increase the overall throughput of the default search, insert and remove operations. To materialize our design, we redesigned the LFHT data structure in order to maintain the lock-freedom property of the existent search, insert, remove and expand operations.

Experimental results show that lock-free compression effectively improves the default operations and, in doing so, it outperforms the previous design, which was already quite competitive when compared against the concurrent hash map design in Intel's TBB library. We argue that our experimental results are very interesting and show the potential of our design since it was able to achieve better throughput ratios than CHM, in almost all scenarios, and, for some thread launches, the difference between the two is very significant. This is quite an accomplishment if we consider that both the CHM design and the hardware architecture are implemented by Intel.

As further work, we plan to extend our design to implement a scheme that allows lock-free compression to be split into several subtasks that can be executed concurrently by different threads, instead of just a single thread as it is now, and compare its performance in different hardware architectures using real-world applications.

References

1. Acharya, A., Zhu, H., Shen, K.: Adaptive algorithms for cache-efficient trie search. In: Selected Papers from the International Workshop on Algorithm Engineering and Experimentation. p. 296–311. ALENEX '99, Springer-Verlag (1999)
2. Areias, M., Rocha, R.: Towards a Lock-Free, Fixed Size and Persistent Hash Map Design. In: International Symposium on Computer Architecture and High Performance Computing. pp. 145–152. IEEE (2017)
3. Bagwell, P.: Ideal Hash Trees. Es Grands Champs **1195** (2001)
4. Brown, T.: Techniques for Constructing Efficient Lock-free Data Structures. Ph.D. thesis, University of Toronto (2017)

5. Evans, J.: Tick tock, malloc needs a clock. In: Applicative 2015. ACM, New York, NY, USA (2015)
6. Fredkin, E.: Trie Memory. Communications of the ACM **3**, 490–499 (1962)
7. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. Journal of Experimental Algorithmics **19** (2015)
8. Herlihy, M., Wing, J.M.: Axioms for Concurrent Objects. In: ACM Symposium on Principles of Programming Languages. pp. 13–26. ACM (1987)
9. Knuth, D.E.: The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison-Wesley Longman (1998)
10. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the Ninth European Conference on Computer Systems. EuroSys '14, ACM (2014)
11. Mehta, D.P., Sahni, S.: Handbook of Data Structures and Applications. Chapman & Hall/CRC (2004)
12. Moreno, P., Areias, M., Rocha, R.: Memory Reclamation Methods for Lock-Free Hash Tries. In: International Symposium on Computer Architecture and High Performance Computing. pp. 188–195. IEEE (2019)
13. Prokopec, A.: Cache-tries: Concurrent lock-free hash tries with constant-time operations. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 137–151. ACM (2018)
14. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent Tries with Efficient Non-Blocking Snapshots. In: ACM Symposium on Principles and Practice of Parallel Programming. pp. 151–160. ACM (2012)
15. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly (2007)
16. Sagiv, Y.: Concurrent operations on B*-trees with overtaking. Journal of Computer and System Sciences **33**(2), 275 – 296 (1986)