Memory Reclamation for an Elastic Lock-Free Hash Trie Map Design Using Hazard Pointers

João Chamiça Pereira, Pedro Moreno, and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto, Portugal
{jpereira,pmoreno,ricroc}@dcc.fc.up.pt

Abstract. A hash map is elastic if it can expand and compress. Hash maps expand in order to reduce collisions and compress in order to reduce depth and memory usage. Starting from a particular elastic lock-free hash map design, which implements expansion and compression in constant time while maintaining the high throughput of lock-freedom, we focus on solving the problem of memory reclamation outside garbage collected environments without losing the lock-freedom property. We propose a lock-free and safe memory reclamation method using hazard pointers that is compatible with the compression mechanism of this data structure. Experiments show that our approach obtains results on par with the best state-of-the-art memory reclamation methods, both in execution time and memory footprint.

1 Introduction

Lock-free data structures [5] ensure that the suspension of one thread will not prevent other threads from progressing, making them suitable for asynchronous systems and for applications which demand low latency like real-time applications. With lock-free data structures the management of locks is unnecessary, which prevents deadlocks, livelocks, and priority inversion [4]. Furthermore, without locks, we can reduce the amount of context switches and waiting queues. In particular, context switches force a processor pipeline to flush, reload cache entries, save processor registers, and the operating system scheduler to execute.

Lock-free programming can make use of distinct synchronization primitives as replacements to mutual exclusion. In particular, the single width CAS (*compare-and-swap*) atomic instruction, available in most processors, is commonly used to implement lock-free data structures. However, its usage introduces an important problem known as the *memory reclamation problem*. In an environment without garbage collection, we cannot directly free the memory of a concurrent object since other threads could still hold a reference to it. A solution is to design a mechanism to determine when an object can be reclaimed. We must, therefore, be able to check if no thread holds a reference to the objects we want to free.

In this work, we focus on solving the problem of memory reclamation outside garbage collected environments for an elastic lock-free hash trie map data structure, named LFHT [9]. We say that a hash map is *elastic* if it can expand and compress. Otherwise, if it cannot compress, we call it *static*. Hash maps expand

in order to reduce collisions and compress in order to reduce depth and memory usage. The LFHT data structure implements expansion and compression in constant time, while maintaining the high throughput of lock-freedom. LFHT's original design was implemented for use under garbage collected environments. More recently, Moreno et al. [10] proposed a novel memory reclamation method for the LFHT data structure, named *Hazard Hash and Level (HHL)*, that explores the characteristics of the LFHT structure in order to achieve efficient memory reclamation with low and well-defined memory bounds. However, the HHL method is not compatible with LFHT's compression mechanism. In this work, we propose a lock-free and safe memory reclamation method based on hazard pointers [3] that is compatible with the compression mechanism of this data structure. Experimental results show that our approach obtains results on par with the HHL method, both in execution time and memory footprint.

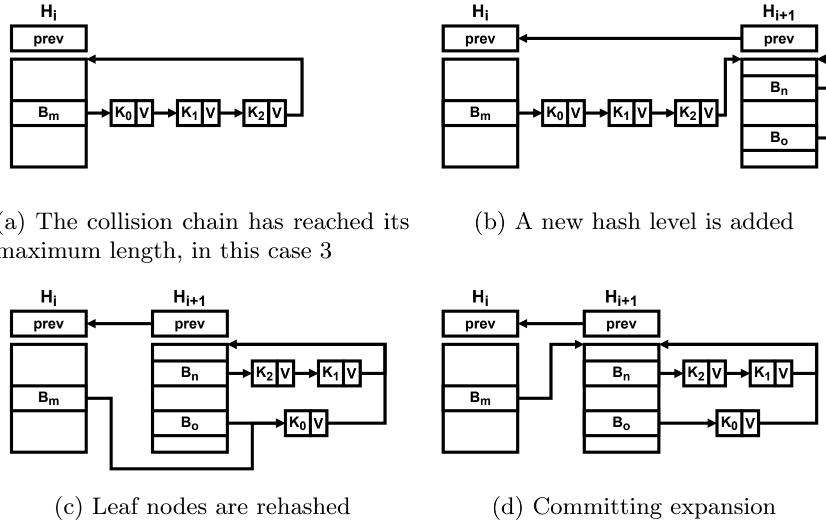
The remainder of the paper is organized as follows. Section 2 describes the LFHT data structure and Section 3 introduces the memory reclamation problem and possible solutions applied to LFHT. Next, Section 4 describes our method for reclaiming memory for LFHT nodes. Finally, Section 5 presents experimental results and then Section 6 concludes the paper by outlining some conclusions.

2 Elastic Lock-free Hash Trie Map

Classic hash maps use a *linear hashing* strategy to convert keys to integers which, in turn, are used to index a monolithic array. On the other hand, hash trie maps use a *partition hashing* strategy, as proposed by Bagwell [1], where a hash is partitioned into chunks of W bits and each chunk is used to index a different array of the map. The map forms a tree hierarchy with arrays of fixed size 2^W. Each array we traverse to belongs to a deeper level of the tree.

A hash trie map has two types of nodes: leaf nodes, and hash nodes. Leaf nodes are key-value pairs while hash nodes are the fixed size arrays which together form a tree. We call each entry of these arrays a *bucket*. When two keys collide on the same bucket, during insertion, the leaves form a singly linked list called a *collision chain*. When the collision chain reaches a certain threshold, it will expand to a new level of the tree. As collision chains get longer, so does the average number of hops per lookup, or *average path length*. Expanding to a new level will reduce the average path length of lookup and, therefore, improve performance. The process of moving nodes to a new level is called *rehashing*. Unlike the classic hash map, instead of rehashing all nodes of the map, only the nodes of a collision chain will be moved to a new array.

Figure 1 illustrates LFHT's expansion procedure. Each node in the collision chain consists of a key-value pair, a reference to the next node in the chain and a flag with the state of the node, which can be valid (V) or invalid (I). The collision chain must be expanded in reverse order, starting from its tail on to the head, one node at a time, to ensure all nodes remain reachable by any thread at any point in time. Thus, first K_2, then K_1, and finally K_0.



(a) The collision chain has reached its maximum length, in this case 3

(b) A new hash level is added

(c) Leaf nodes are rehashed

(d) Committing expansion

Fig. 1: LFHT's expansion procedure

To remove nodes we follow a two step process as proposed by Harris [2]. This process is illustrated in Fig. 2. First, the node we wish to remove is marked as invalid, K_2 in Fig. 2, effectively making it immutable. Then, we detach it from the collision chain by updating the pointer of the previous valid node on the collision chain to the node in front of our target. The first step is necessary to prevent the insertion of nodes in front of already detached ones. Any change to the tree, when adding or removing leaf or hash nodes, will require a CAS operation. For example, setting one node with a reference to another node requires a CAS.

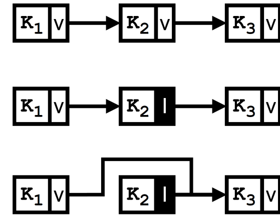
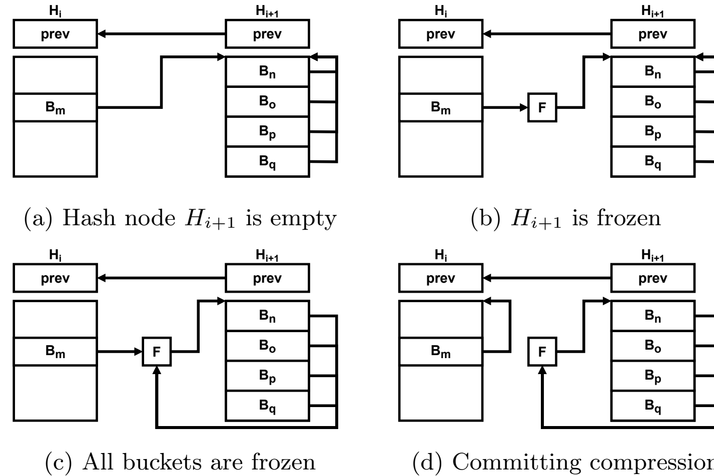


Fig. 2: LFHT's leaf removal

In a process called *compression*, a hash node is removed if all its buckets become empty after removing nodes. The procedure starts by placing a special *freeze* node in between the target hash node (the one we wish to remove) and its parent. Then we try to set every bucket to point to the *freeze* node, one by one. If this process fails, then it must be because a new node was inserted concurrently, and we must revert all changes of the compression. Figure 3 illustrates the compression procedure. After successfully compressing one hash node, we could attempt to compress its parent.

3 Memory Reclamation Problem

Outside garbage collected environments, we may not be able to free memory right after removal of nodes from the LFHT data structure. For example, after



(a) Hash node H_{i+1} is empty (b) H_{i+1} is frozen

(c) All buckets are frozen (d) Committing compression

Fig. 3: LFHT's compression procedure

removing a leaf node N, we are not sure whether some other thread has a local reference to N. In the worst case, if we free the memory of N, we risk a *use after free* problem like the following:

1. Thread T_1 stops amidst of traversing through node N_1.
2. T_2 removes and frees the memory of N_1.
3. T_2 allocates a new node N_2 calling *malloc*. The reference given by the allocator points to the exact same memory block of the previously deleted node N_1. N_2 is inserted in a completely different place on the hash map.
4. T_1 traverses through N_2, which has the same reference as the deleted N_1. T_1 has moved to another location in the structure.

Hazard pointers [3] is a memory reclamation method that solves the memory reclamation problem. In this method, each thread maintains a number of single-writer multiple-reader pointers, called hazard pointers, which are used to prevent objects from having their memory reclaimed incorrectly. When moving through a data structure, a thread sets its hazard pointers to the current objects being accessed. This action is called *protection* of an object. Later, if another thread removes an object, it starts by placing it into a reclamation list and only when this list reaches a certain threshold, it executes the reclamation procedure trying to free the objects stored in the reclamation list. An object may have its memory freed if and only if it is not protected by any hazard pointer.

Hazard pointer protection comes at a high cost. Protection requires the use of *sequentially consistent writes*, which will force the thread's write buffers to flush. Furthermore, to prevent race conditions, after setting a hazard pointer, we must check again if the reference we just protected is still valid in the data structure. This additional step is called *protection safety check*. These costly

overheads can be mitigated by using *asymmetric barriers* [7], which provide protection by moving the cost of memory management from the principal code path to the less frequent memory reclamation procedure, significantly reducing or eliminating memory barriers executed on the principal code path. As a result, the reclamation procedure will be slower, but since it is rarely executed, we achieve a positive net performance improvement.

The *Hazard Hash and Level (HHL)* method [10] is a specific memory reclamation method designed for the LFHT data structure, which is less costly than hazard pointers, since often only two sequentially consistent writes are required per lookup operation. The name HHL is due to the approach of protecting entire collision chains of leaf nodes using *hazard pairs*. A hash-level pair uniquely identifies a collision chain. The hash of a key refers to the path where the collision chain is, whereas the level of the hierarchy selects a portion of the path. To protect a pair, two separate writes are needed. However, when the HHL was proposed, the LFHT structure did not support compression and hence the HHL method was not designed with this in mind.

4 Our Contribution

To mitigate HHL's lack of support for LFHT's compression, we propose a lock-free and safe memory reclamation method based on hazard pointers that is compatible with LFHT's compression mechanism. In what follows, we discuss in detail our contribution. The key goals of our approach were:

- Fully support compression, which includes the ability to reclaim memory from removed hash nodes and removed leaf nodes
- Maintain the lock-freedom property
- Guarantee well-defined memory bounds

In our approach, a hazard pointer can be used to protect the individual reference of either a hash node or a leaf node. Before traversing through a hash or leaf node, we thus protect its reference and perform a safety check. Two hazard pointers are used to protect pairs of subsequent hash nodes while the other hazard pointers are used to protect the leaf nodes in the collision chains.

Figure 4 illustrates the protection of nodes during the lookup operation. It considers that a thread $T1$ is traversing the collision chain starting from bucket B_p and managed to protect the hash nodes H_{i+2} and H_{i+3} and the leaf nodes K_0 and K_1 and is now trying to follow the chain from K_1 to K_2. Consider also that, concurrently, another thread $T2$ removed K_1 and K_2 from the hash map. We know that K_1 will not have its memory reclaimed because it is protected by $T1$. However, since K_2 was not protected before its removal, it could have been already freed. Hence, to not risk accessing the potentially freed memory block of K_2, $T1$ must restart traversing from bucket B_p.

Algorithm 1 summarizes the use of hazard pointers for the *lookup()* procedure, given a hash node H and a hash h. At the start, we determine the *index* of the bucket at level $H.level$, using bit-wise shifts and masks to select a chunk of

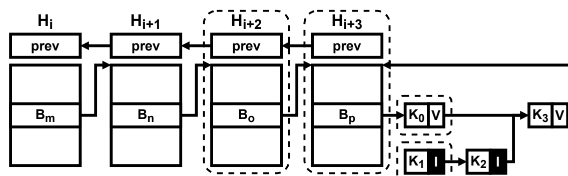


Fig. 4: Protecting nodes during lookup (dotted lines represent HP protections)

W bits from the hash h (line 1). Then, we read the bucket $H[index]$ and try to protect its reference (lines 2–3). The *HpProtect()* procedure sets the reference of the first argument to one of the thread's hazard pointers and performs the protection safety check by re-reading the second argument. If the protection safety check fails, we should restart the algorithm from the beginning (line 4).

Next, if the bucket points to a freeze node, we try to skip it and move to the next node (lines 5–11). In the continuation, if the freeze node or the bucket point to a new hash node, we traverse down a level of the tree (lines 14–15). Otherwise, we traverse through the collision chain (lines 16–25). We must always try to protect the reference of the next node (lines 23–24) before moving to it (line 25) and, if we find an invalid leaf node, we must attempt to remove it using the *Remove()* procedure (line 17). If the removal is successful, the removed node goes to the thread's reclamation queue and we may proceed. If the removal fails, we must restart lookup (line 18). Finally, if we find the node with the given hash h, we return successfully (lines 21–22).

Algorithm 1 Lookup(H, h)

1: $index \leftarrow GetChunk(h, H.level, W)$
2: $iter \leftarrow H[index]$
3: **if** $\neg$HpProtect($iter, H[index]$) **then**
4:     **return** Lookup(H, h)
5: **if** $iter$ is a freeze node **then**
6:     $nxt \leftarrow iter.next$
7:     **if** $\neg$HpProtect($nxt, iter.next$) **then**
8:         **return** Lookup(H, h)
9:     **if** $nxt = H$ **then**
10:         **return** Lookup($RootH, h$)
11:     $iter \leftarrow nxt$
12: $prev \leftarrow H$
13: **while** $iter \neq H$ **do**
14:     **if** $iter$ is a hash node **then**
15:         **return** Lookup($iter, h$)
16:     $nxt \leftarrow iter.next$
17:     **if** $iter$ is invalid and $\neg$Remove($iter, h$) **then**
18:         **return** Lookup(H, h)
19:     **else if** $iter$ is valid **then**
20:         $prev \leftarrow iter$
21:         **if** $iter.hash = h$ **then**
22:             **return** TRUE
23:     **if** $\neg$HpProtect($nxt, prev.next$) **then**
24:         **return** Lookup(H, h)
25:     $iter \leftarrow nxt$
26: **return** FALSE

As mentioned above, each thread may require two hazard pointers, to protect pairs of subsequent hash nodes, plus as many hazard pointers as the maximum length of a collision chain, to protect leaf nodes. In what follows, we give more details about the reason for these numbers.

First, why do we need to protect more than one hash node at a time? After setting a hazard pointer to protect a hash node, the node can still be removed by another thread just before we are able to protect it. To prevent this, we must perform an additional protection safety check to verify if the protection was successful and, if the safety check fails, we must restart the operation. This requires one hazard pointer to protect the parent hash node and another to protect the child. This is necessary to prevent the parent hash node from having its memory reclaimed before the additional safety check. Thus, to traverse any list or tree, we need at least two hazard pointers.

For the leaf nodes, however, due to the expansion procedure, we must protect every node of the collision chain with a separate hazard pointer. This is necessary because, when rehashing nodes to a new level, we do that by moving from the tail to the head of the collision chain but, since leaf nodes lack a reference to the previous node on the collision chain, we cannot protect leaf nodes during expansion and we must protect all nodes before starting the expansion.

During lookup, a thread may need to traverse an infinite amount of leaf nodes. For example, if the previous leaf node of a collision chain is removed and a new node is added to the front, the thread will exhaust all its hazard pointers. To solve this issue, we must count the number of hops when traversing collision chains. If the number of hops exceeds the maximum length of a collision chain, we must go back to the beginning of the chain.[1]

In conclusion, we need at most $2+L$ hazard pointers per thread (L being the maximum length of the collision chain). Bounded memory is guaranteed because only at most $(2+L)\times NT$ references (NT being the maximum number of threads in execution) can be protected at any instant.

4.1 Other Important Changes

To completely ensure the correctness of our approach, the following changes were also applied (we discuss each one in more detail next):

- Threads must remove invalid nodes during lookup
- Threads must cooperate in an ongoing expansion before proceeding
- After compression, the *prev* field of a hash node must be invalidated

In Fig. 4, we have already illustrated an example where a thread needs to restart traversing from the head of a collision chain if an invalid node N is found since, otherwise, it would not be possible to protect the next node on the chain. However, if the thread responsible for removing N is blocked for long, other threads will continuously loop back to the head of the chain. Therefore, to prevent obstruction, threads must detach invalid nodes from the collision chain.

During expansion, leaf nodes are moved to a new level L. If other threads insert new nodes in L concurrently, the thread responsible for expanding may need to protect these newly inserted nodes but may not have more hazard pointers available. This situation is illustrated in Fig. 5. Dotted lines represent the

[1] For the sake of simplicity, this is omitted in Algorithm 1

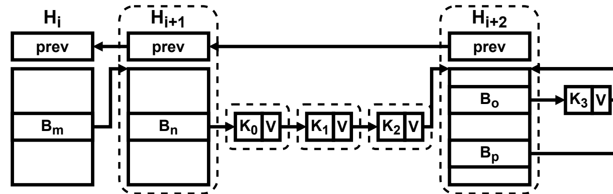


Fig. 5: Not enough hazard pointers when expanding

nodes protected with hazard pointers by the expanding thread T and node K_3 represents a node inserted by another thread. Because T has no more available hazard pointers, it would need to discard one already in use in order to protect K_3, which may not be possible. To prevent this kind of situation, threads cooperate in an ongoing expansion, before adding new nodes. With this, K_3 would never be inserted before all nodes were rehashed.

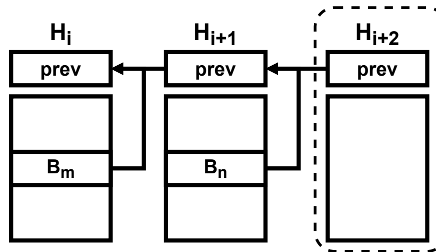
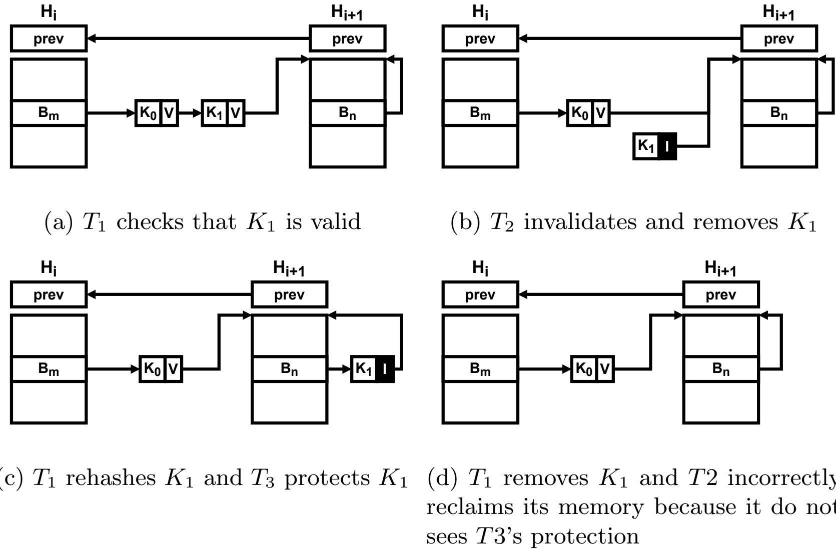


Fig. 6: Unable to protect a parent hash node

Every hash node has a reference to its parent in a field called *prev*. This field is an important part of the compression mechanism since, when a hash node is successfully compressed, we follow *prev*'s reference to also attempt to compress the parent hash node. However, since the *prev* field never changes its value, the hazard pointer safety check is useless. The *prev* field will still point to the parent even after the parent has been reclaimed. In Fig. 6, thread T compresses H_{i+2} and then tries to protect H_{i+1} before starting its compression. However, in the meantime, some other thread completely removes H_{i+1} and frees its memory. Because the *prev* field of H_{i+2} remains unchanged, T cannot use it as a way to protect H_{i+1}. To make it possible to protect the parent hash node from the *prev* field, we must therefore change its value to *null* before removing the child hash node. And before modifying it to *null*, we protect the parent hash node.

4.2 Delegation Problem

During expansion, if a thread $T1$ only sees a node N as invalid after moving it to the next hash level (because concurrently another thread $T2$ marked N as invalid), $T1$ will become responsible for making N unreachable. The process of transferring this responsibility to the expanding thread is called *delegation*. One flaw with this process, called the *delegation problem*, is illustrated in Fig. 7.



(a) T_1 checks that K_1 is valid

(b) T_2 invalidates and removes K_1

(c) T_1 rehashes K_1 and T_3 protects K_1

(d) T_1 removes K_1 and $T2$ incorrectly reclaims its memory because it do not sees $T3$'s protection

Fig. 7: LFHT's delegation problem

The solution implemented in the HHL method for the delegation problem is to check all hazard pointers twice during the reclamation procedure [10]. In our approach, the delegation problem only occurs for nodes at the end of an expanding collision chain and we can prevent the delegation problem altogether by performing one additional check before invalidating a node. We check whether its next neighbor is a new level and, in such cases, we must aid the expansion procedure and only then we can remove the node. With this, we avoid loading all hazard pointers twice during the reclamation procedure.

5 Experimental Results

Our experiments were run on a NUMA machine with 4 AMD Opteron 8425 HE CPUs with 6 cores each. This causes a latency penalty when cores from different nodes have to synchronize, so, when we increase the number of cores to 7, 13, and 19, the performance may degrade slightly.

We compare 5 different implementations of the LFHT hash map: (i) SNF (static no free), which does not implement compression and does not reclaim memory; (ii) ENF (elastic no free), with compression but no memory reclamation; (iii) EHP (elastic with hazard pointers), our elastic version with memory reclamation based on hazard pointers; (iv) EHPA (elastic with hazard pointers and asymmetric barriers), also our approach based on hazard pointers but using asymmetric memory barriers, as described in section 3; (v) HHL, a static implementation using the hazard hash and level memory reclamation method. Our goal is to compare our EHP and EHPA implementations against the others.

For benchmarking, we used a specific program that measures, along with various metrics, the time it takes for multiple threads to perform N hash map operations. Each thread will generate keys using a pseudo random number generator for the hash map operations, which can be either insertions, removals, or searches. Figure 8 illustrates the benchmark program in all its phases for N map operations with 25% insert operations, 25% remove operations, 25% successful search operations and 25% failed search operations (respectively, group columns named Insert, Remove, Fetch and Miss in Fig. 8).

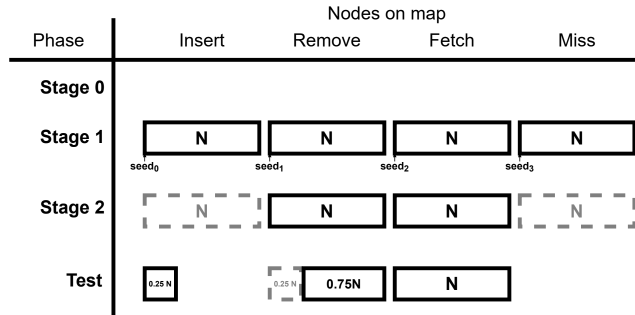


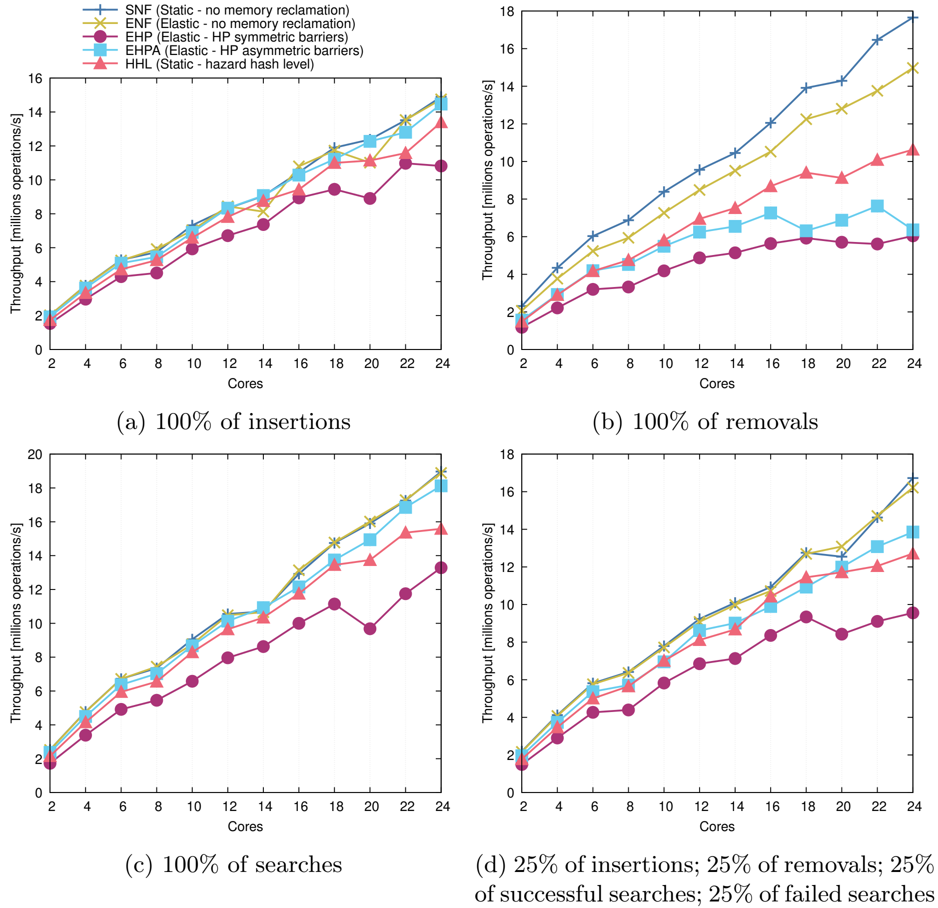
Fig. 8: Benchmark program phases

Before starting the benchmark, the hash map is filled with a number of nodes in a two stage process. The first stage adds N unique keys to each group of operations ($4N$ keys in total), with each group of generated keys starting with a different seed. The second stage removes the $2N$ keys for the Insert and Miss groups of operations. By preemptively removing a number of keys, we simulate the scenario where the map has been in use for a long period of time. This allows us to more accurately show the advantage of the elastic version as it benefits from a lower number of hash nodes and shorter average path length. Moreover, this also guarantees that, during the test phase, each removal and search operations will indeed target existing keys in the map while insertion operations will add new keys.

For the benchmark results presented next we used a chunk size of 4, which corresponds to hash nodes with 2^4 buckets, a collision chain length of 4 and a reclamation frequency of 10,000 nodes, which means that the reclamation procedure is triggered for every 10,000 nodes removed for the hash map and stored in the reclamation list. Figure 9 shows the benchmark results obtained for 2^{24} operations and an average of 20 samples for each measurement of throughput from 2 to 24 threads (cores).

On average, HHL applies two protections per lookup, a contrast to our EHP method which protects every node traversed through. Each protection is done with a sequentially consistent write. In order to mitigate this issue, the EHPA method takes advantage of the system call *mem_barrier*, which implements an asymmetric barrier. With this function, every hazard pointer protection has the lowest memory ordering constraints, as opposed to sequential consistency, although a compiler barrier must still be used.

As expected, the removal benchmark (Fig. 9b) shows a significant slowdown of the elastic versions, especially the ones using hazard pointers. However, this particular test case using only removal operations is rare. In most cases, the



(a) 100% of insertions

(b) 100% of removals

(c) 100% of searches

(d) 25% of insertions; 25% of removals; 25% of successful searches; 25% of failed searches

Fig. 9: Benchmark results

number of search operations outnumbers insertions and removals, and a certain number of removals implies that at least the same number of insertions occurred previously. On the other hand, when considering only insertions or searches, as shown in Fig. 9a and 9c respectively, the EHPA implementation is clearly faster when compared to both EHP and HHL. Finally, Fig. 9d shows the case of mostly performing searches (25% of successful searches and 25% of failed searches) with simultaneously inserting and removing nodes. The results show that for this more general case, EHPA can again outperform both EHP and HHL, which is a fantastic result as HHL does not implement elasticity. On top of having similar results in throughput to HHL, the EHPA implementation also reclaims the memory of hash nodes. Thus, more memory can be reclaimed throughout the program's execution in comparison to HHL.

6 Conclusion

We have proposed a memory reclamation method for a lock-free hash map data structure named LFHT. To the best of our knowledge, this is the first implementation of a memory reclamation method compatible with LFHT's compression operation, being thus able to reclaim memory for both types of nodes, be they key-value pair nodes or internal tree hash nodes. On top of having lower memory footprint, our design showed similar throughput when compared to the best state-of-the-art method (*HHL*), and can even outperform it in some scenarios.

For future work, we plan to extend our reclamation method to support back expansion. We also plan to compare our memory reclamation method against the *Automatic Optimistic Access* [6] and the *Free Access* [8] methods.

Acknowledgments

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

References

[1] P. Bagwell. *Ideal hash trees.* Tech. rep. 2001.

[2] T. Harris. "A pragmatic implementation of non-blocking linked-lists". In: *International Symposium on Distributed Computing.* Springer. 2001, pp. 300–314.

[3] M. Michael. "Hazard pointers: Safe memory reclamation for lock-free objects". In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (2004), pp. 491–504.

[4] K. Fraser and T. Harris. "Concurrent programming without locks". In: *ACM Transactions on Computer Systems* 25.2 (2007), 5–es.

[5] M. Herlihy and N. Shavit. "On the nature of progress". In: *International Conference On Principles Of Distributed Systems.* Springer. 2011, pp. 313–328.

[6] N. Cohen and E. Petrank. "Automatic memory reclamation for lock-free data structures". In: *ACM SIGPLAN Notices* 50.10 (2015), pp. 260–279.

[7] D. Dice, M. Herlihy, and A. Kogan. "Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures". In: *International Symposium on Memory Management.* 2016, pp. 36–45.

[8] N. Cohen. "Every data structure deserves lock-free memory reclamation". In: *ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–24.

[9] M. Areias and R. Rocha. "Towards an Elastic Lock-Free Hash Trie Design". In: *International Symposium on Parallel and Distributed Computing.* Cluj-Napoca, Romania (online event): IEEE Computer Society, 2021, pp. –.

[10] P. Moreno, M. Areias, and R. Rocha. "On the Implementation of Memory Reclamation Methods in a Lock-Free Hash Trie Design". In: *Journal of Parallel and Distributed Computing* 155 (2021), pp. 1–13.