Memory Reclamation Methods for Lock-Free Hash Tries

Pedro Moreno and 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: {pmoreno,miguel-areias,ricroc}@dcc.fc.up.pt

***Abstract*—Hash tries are a trie-based data structure with nearly ideal characteristics for the implementation of hash maps. Starting from a particular lock-free hash map data structure, named *Lock-Free Hash Tries (LFHT)*, we focus on solving the problem of memory reclamation without losing the lock-freedom property. We propose an approach that explores the characteristics of the LFHT structure in order to achieve efficient memory reclamation with low and well-defined memory bounds. Experimental results show that our approach obtains better results when compared with other state-of-the-art memory reclamation methods and provides a competitive and scalable hash map implementation, if compared to lock-based implementations.**

***Index Terms*—Memory Reclamation, Lock-Freedom, Hash Maps, Hazard Pointers.**

I. Introduction

The traditional approach to synchronize access to critical sections is to use blocking primitives, such as, mutexes or semaphores. An algorithm is *non-blocking* if failure or suspension of any thread cannot cause failure or suspension of another thread. In general, non-blocking algorithms use atomic read-modify-write primitives, the most notable of which is the CAS instruction. A non-blocking algorithm is also *lock-free* if there is guaranteed system-wide progress, i.e., there is no per-thread progress guarantee (individual threads can starve) but it is guaranteed that at least one thread progresses in some well-defined number of steps, regardless of the scheduling policy.

There are multiple implementations of lock-free data structures, but most of them are not entirely usable in a lock-free manner, as they delegate the task of *memory reclamation* to a garbage collector. This is a problem as it avoids portability to environments where a garbage collector is not available or, if available, it is not lock-free. This leads to the loss of the overall lock-freedom property, as one of the pieces does not have the property. On the other hand, many memory reclamation schemes were developed for general lock-free data structures. However, some are not lock-free themselves [1]–[3] or not compatible with all lock-free data structures [4]–[7].

The memory reclamation of removed elements on a lock-free data structure is not as simple as in a lock-based one. To ensure lock-freedom, we need to allow concurrent accesses to the elements on the data structure and, as such, we cannot guarantee that an element is not being accessed by other threads at the moment it is removed. Overcoming this limitation requires sophisticated methods to postpone, delegate and determine when the reclamation may occur. These methods should also offer some guarantees on performance and memory usage bounds while keeping the lock-freedom property. For example, if the memory reclamation method is not able to reclaim the removed elements (i.e., progress) at the same rate as they are removed from the data structure, we may end up with unbounded memory consumption. In this work, we focus on extending a sophisticated implementation of a lock-free hash map data structure, named *Lock-Free Hash Tries* (*LFHT*) [8], to support efficient memory reclamation in a lock-free manner. The LFHT implements a hash map that settles for a hierarchy of hash tables instead of a monolithic expanding one, granting it good latency and throughput characteristics. Additionally, we propose a novel method based on the idea of using a *hazard hash* and a *hazard level* to represent, respectively, a path and a level in LFHT's hierarchy. This results in a small and well-defined portion of memory being protected from reclamation by every thread, and in fewer updates done on such hazard pairs during an operation. The resulting lock-free memory reclamation method, which we named *Hazard Hash and Level* (*HHL*), achieves lower synchronization overhead than any of the state-of-the-art lock-free memory reclamation methods, while providing very well-defined and flexible memory bounds.

Experimental results show that LFHT with the HHL method is able to achieve performance and scalability results surpassing current lock-based implementations, such as the concurrent hash map design in Intel's TBB library [9]. We also show that the current state-of-the-art based methods cause extreme performance degradation on high throughput data structures, such as LFHT. This is caused by their inherent need for updating global information accessed frequently by all threads, which is not the case in the HHL method.

This work is funded by Project 9471 – Reforçar a Investigação, o Desenvolvimento Tecnológico e a Inovação – and by the European Regional Development Fund (ERDF) within project POCI-01-0145-FEDER-016844, and by National Funds through the Portuguese Foundation for Science and Technology (FCT) within project UID/EEA/50014/2019. Miguel Areias is funded by the FCT grant SFRH/BPD/108018/2015.

978-1-7281-4194-7/19/$31.00 ©2019 IEEE

II. Background

To motivate for the problem of lock-free memory reclamation, we start by introducing, as an example, Harris' lock-free linked list [10] for storing key/value pairs that is implemented as follows. Each node in the list consists of a key, a value associated with the key, 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 flag is considered to be embedded in the next node reference (often the least significant bit of the reference, as it does not store any information due to memory alignment). The list begins with a special header node H, which references the first node on the list. To mark the end of the list, the last node references back to the header node H.

During its lifetime, a node can be in one of the following states: *valid*, *invalid*, *unreachable* or *reclaimable*. Fig. 1 shows a possible configuration illustrating these states. Node $K1$ is considered valid because it is reachable from the header node H and its flag is set to valid (V). Node $K3$ is considered invalid since it is reachable but its flag is set to invalid (I). Finally, node $K2$ is considered unreachable (i.e., logically removed from the list) since it is not reachable from H. An unreachable node is always an invalid node. Despite node $K2$ being unreachable, some threads may still have local references to it, as a result of reaching $K2$ before it was made unreachable. When it is determined that there are no longer references to an unreachable node by any thread, then it is safe to physically reclaim the node's memory. A node in this state is considered reclaimable.

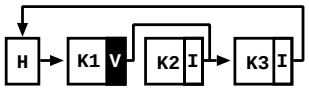


Fig. 1: Node states

Fig. 2 shows how the remove operation changes a node state. First, the node is made invalid by changing its flag from V to I, which informs the other threads that the node is being removed from the list. Next, the node is logically removed from the list by updating the corresponding next on chain reference of the previous valid node, which makes the node unreachable for the upcoming threads accessing the list. After these two steps, the node is considered removed and the remove operation ends by adding it to a local/global reclamation queue. In general, the reclamation procedure begins when the amount of nodes in a reclamation queue reaches a threshold. That threshold can be tuned in order to exchange memory usage by execution time.

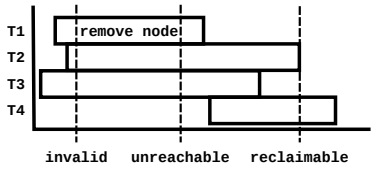


Fig. 2: Node states after removal

The problem arises in deciding when a node in a reclamation queue becomes reclaimable, i.e., when no thread has a reference to it. This can happen after the node was made unreachable and until the end of the last operation, that started before the node was made unreachable, finishes. From this point forward, it is impossible for any other thread to have a reference to the node, making it certain to be reclaimable after this point. This is the case of thread $T4$ in Fig. 2, which started its operation after the node was made unreachable.

There are two main methodologies to determine if a node can be reclaimed, one is to use the order of events in time, another is to track the spacial location of threads. For time based methods, a *quiescent state* is a moment in time where a thread has no access to shared resources and a *grace period* is a period of time in which all threads have been in at least one quiescent state. When a node is made unreachable, we can be certain that, after a grace period has elapsed, no thread still references it. Based on this idea, some well-know methods to establish relative temporal orders between events are the usage of *global epochs* [1] or *Lamport clocks* [11]. A general problem with these methods is that the starvation of a single thread prevents the occurrence of grace periods, which can prevent unbounded amounts of memory from being reclaimed. In space based methods, *hazard pointers* [5] are shared variables that hold pointers to shared resources currently in use by a thread. They are used to inform the other threads of what data structures are being accessed by a thread and that thus cannot be reclaimed by others. Hazard pointers usually imply a significant overhead caused by threads having to share their location every time they move on the data structure. More recent methods, such as *Hazard Eras* [7] and *Drop the Anchor* [6], use mixed methodologies to guarantee bounds on the amount of irreclaimable memory and low overheads.

III. 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. Fig. 3 shows how the insertion of nodes is done in a hash level.

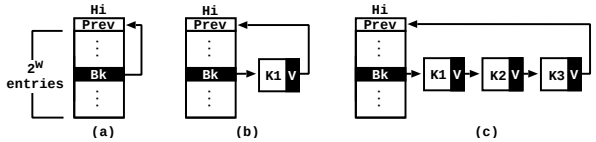


Fig. 3: Insertion of nodes in a hash level

Fig. 3a shows the initial configuration for a hash level. Each hash level is formed by a hash node H_i, which includes a bucket array of 2^w entries and a backward reference *Prev* to the previous hash level, and by the corresponding chain of nodes per bucket entry. Initially, all bucket entries are empty. In Fig. 3, 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. Fig. 3b shows the configuration after the insertion of node $K1$ on B_k and Fig. 3c shows the configuration after the insertion of nodes $K2$ and $K3$. 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).

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. Fig. 4 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} and the *Prev* field referencing H_i (as shown in Fig. 4a). From this point on, new insertions will be done on the new level H_{i+1} and the chain of leaf nodes on H_i will be moved, one at a time, to H_{i+1}. Fig. 4b and Fig. 4c show how node $K3$ is first mapped in H_{i+1} (bucket B_n) and then moved from H_i (bucket B_k). It also shows a new node $K4$ being inserted simultaneously by another thread. When the last node is expanded, the bucket entry in H_i references H_{i+1} and becomes immutable (Fig. 4d). Immutable fields are represented with a white background.

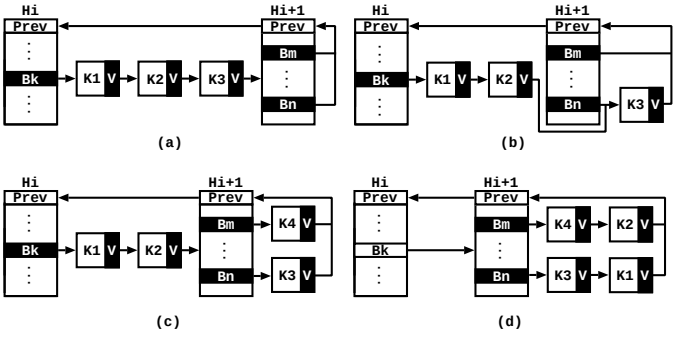


Fig. 4: Expansion of nodes in a hash level

Next, Fig. 5 shows an example illustrating how a node is removed from a chain. The remove operation can be divided in two steps: (i) the invalidation of the node (shown in Fig. 5a) and (ii) making the node unreachable (shown in Fig. 5b). The invalidation step starts by finding the node N we want to remove and by changing its flag from valid (V) to invalid (I). If the flag is already invalid, it means that another thread is also removing the node and, in such case, nothing else needs to be done. Next, to make the node unreachable, first we need to find the next valid node A on the chain (note that it can be the hash node corresponding to the level N is at). Then, we continue traversing the chain until we find a hash node H (if we have not yet). If H is the same hash node we have started from, we traverse again the chain until we find the last valid node B before N (or we consider the bucket entry if no valid node exists). If, while searching for B we do not find node N, it means that N has already been made unreachable and our job is done. Otherwise, we just need to change the reference of B to A. This is shown in Fig. 5b, where $K1$ refers to $K3$.

If H is not the same hash node we have started from,

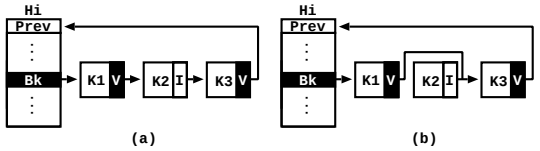


Fig. 5: Removal of nodes in a hash level

this means that a concurrent expansion is happening simultaneously and we restart the process in the next level (note that node N could either have been expanded before we have invalidated it or is currently in the process of being expanded). In the case N has been expanded before we made it invalid, we will be able to make it unreachable in the next level. Otherwise, if N is in the process of being expanded, we do not need to make it unreachable, as the expanding thread will not expand it or will make it unreachable, if it only sees N as invalid after completing its expansion. In this situation, the thread doing the expansion becomes responsible for making the node unreachable. The process of transferring this responsibility to the expanding thread is called *delegation*.

IV. Problem Definition & Challenges

By default, all the state-of-the-art memory reclamation methods rely on the fact that an element being removed from a data structure is left in an unreachable state when the remove operation terminates. However, in the original design of the LFHT data structure, a node is not guaranteed to be unreachable at the end of the remove operation, if a concurrent expansion is happening simultaneously and the task of making the node unreachable was *delegated* to the expanding thread.

Fig. 6 illustrates how an expansion operation can change the moment where a node is considered unreachable and reclaimable. In particular, the assumption that a thread starting after the end of the remove operation cannot have a reference to the removed node is not valid anymore. This is the case of thread $T4$ in Fig. 6, which started its operation after thread $T1$ finished the remove operation, but before the node was made unreachable by the expanding thread $T2$. In this scenario, a node can become reclaimable later than what would be expected if no delegation happened. Avoiding this delegation mechanism is not possible since $T2$ can always reinsert the node in the new hash level before realizing that it was marked as invalid and made unreachable. Fig. 7 illustrates this situation in more detail.

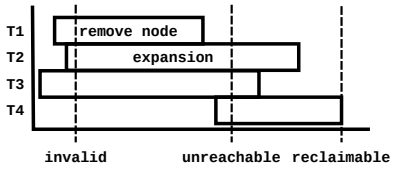


Fig. 6: Node states during expansion

The problem resides exclusively in the case where a thread $T1$, doing an expansion, reads a valid node $K3$ and, before changing the corresponding bucket reference in the new level H_{i+1} in order to expand $K3$ (Fig. 7a), another thread $T2$ is able to invalidate $K3$ (Fig. 7b) and make it unreachable (Fig. 7c). As the removing thread $T2$ does not interfere with the reference in H_{i+1}, the expanding thread $T1$ can succeed

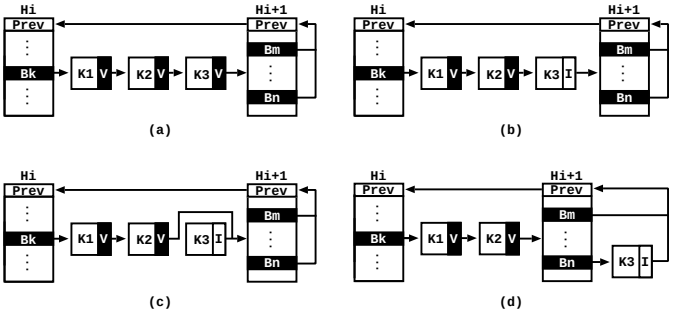


Fig. 7: Reinsertion of an invalid node during expansion

in updating the bucket reference B_n in H_{i+1} to $K3$ and effectively reinsert $K3$ making it reachable again (Fig. 7d).

To apply the state-of-the-art reclamation methods to LFHT we need to avoid the problem and guarantee that a node becomes (permanently) unreachable within the execution of the corresponding remove operation. Our approach was to change the remove operation when a node N is being marked as invalid in a chain that is being expanded (i.e., before making N unreachable). The idea is to search for the spot where N would be expanded to and mark that spot with a special tag. That tag would cause the CAS done by the expanding thread to fail and thus avoid N from being reinserted. The expanding thread would then verify that N was made invalid in the meantime and skip its expansion. This method was implemented and tested extensively without showing any wrong results. However, there is a critical flaw that, under very specific circumstances, can lead to nodes being reinserted after being made unreachable. If multiple expansions occur simultaneously in different hash levels of the same path, they can be trying to expand different nodes into the same point and thus overflow the tag and make the reinsertion of an invalid node possible again. This tag overflow reflects what is known as an ABA problem [12]. Since it is unlikely to happen in practice, and this solution to the delegation problem does not affect the performance of the data structure, we still used this method for benchmarking purposes.

V. Hazard Hash and Level Approach

Hazard pointers have good memory bounds in memory reclamation, however they rely on thread synchronization based in performing sequentially consistent atomic writes on every node being traversed. Reducing this synchronization overhead, while keeping good memory bounds, is a difficult task and, to the best of our knowledge, there is not a good way to merge nodes in well-defined groups and protect them with a single hazard pointer. An interesting characteristic of LFHT is that leaf nodes are already grouped in chains that have a well-defined maximum size. Thus, instead of having a single hazard reference to protect a single node, we have designed a novel approach, named *Hazard Hash and Level (HHL)*, that is able to protect a well-defined group of leaf nodes. In this novel approach, each thread maintains a special *hazard pair* *<HH, HL>*, formed by a *Hazard Hash (HH)* and a *Hazard Level (HL)*, to indicate in which part of the data structure it is positioned. HH represents a path in LFHT and HL represents a portion of this path.

To implement the HHL approach, we had to extend the original LFHT's algorithms and data structures to ensure that a thread cannot have access to nodes outside the portion of the path defined by its current hazard pair. We now ensure the following properties: (i) threads recovering from preemption must progress to a valid data structure (hash node or leaf node) within the same path; and (ii) no new nodes are inserted in a path with an expansion in course. In the original design, threads can be moved to a different path and recover by moving in that path. In the new design, if a thread is moved to a different path, it now returns immediately to the last known hash node and recovers from that point. Also, in the original LFHT design, the insert and expand operations have the same priority, which means that they could be performed concurrently in the same path. In the new design, it is given a higher priority to the expand operation, such that threads must collaborate to finish the undergoing expansions in a path, before inserting new nodes. To implement these properties, the following changes were made to the LFHT data structure: (i) a bucket entry now includes a *hash flag* to indicate if it stores a reference to a next level hash (the hash flag is part of the atomic field that includes the reference); and (ii) a leaf node now includes a *generation field*, indicating the hash level where it was first inserted, and a *level tag*, indicating the hash level where it is at the moment (the level tag is part of the atomic field that also includes the validity flag and the reference to the next-on-chain node). This means that the state information of a leaf node is now given by a generation field G_i and by an atomic tuple with three arguments *<NextNode, LevelTag, ValFlag>*. For example, in Fig. 8c, the value of the generation field for node $K1$ is $G1$, meaning that it was inserted in the hash level $H1$, and the value of the atomic tuple is *<K3, 2, V>*, meaning that it is referring to node $K3$ (1st argument), it is in the hash level $H2$ (2nd argument) and it holds a valid key (3rd argument).

Next, we describe the key ideas behind the HHL approach. In a nutshell, it is based on the fact that each thread executing on the LFHT data structure protects from reclamation a single and well-defined chain of leaf nodes. Therefore, a leaf node N can only be reclaimed if: (i) N is not in a protected chain; and (ii) N has never been there in the past, as a thread could have *seen* it there and still have a reference to it, despite the fact that, in the meantime, N could have been expanded to a deeper level. As discussed before, a node N being removed is added to the thread's local reclamation queue at the end of the corresponding remove operation. We know that N was invalidated during the remove operation, but we do not know if it was made unreachable, since this process could have been delegated to a thread doing a concurrent expansion. A delayed delegation can further postpone the moment where N can be considered reclaimable. To guarantee that the reclamation of N is safe, the following information is required: (i) the hash value corresponding to the key stored in N, which defines the path where N could have been; (ii) the generation field,

which defines the entry point in that path; and (iii) the level tag, that becomes immutable when N is invalidated and thus defines the last hash level where N was in. The reclamation process is then triggered when a local queue reaches a predefined threshold number of nodes. The reclamation procedure begins by reading the list of hazard pairs of all threads and by copying them in a local data structure, much like as in the hazard pointers method. However, for the HHL method, this reading needs to be done twice and use the two copies of the hazard pairs before performing any reclamation of memory for a node. With only one read we cannot avoid the situation where a thread $T1$ is not protecting N, when its hazard pair is read, and then $T1$ accesses N before a second thread $T2$, performing the delegation process, turns N unreachable. If the hazard pair for $T2$ is read next, then it can happen that $T2$ is not protecting N either. The second read of the list of hazard pairs solves the problem because N is now unreachable and thus the previous situation cannot happen again. After reading twice the list of hazard pairs, a node N in the reclamation queue cannot be reclaimed if it is protected by any hazard pair <*HH, HL*>, i.e., if HH equals the hash value of N up to the hazard level HL and if HL is between the generation and the level tag of N. If such hazard pair exists, then the node is kept in the reclamation queue. Otherwise, the thread removes the node from its local queue and reclaims its memory.

Finally, we discuss the safe traversal of nodes in the HHL approach. To guarantee that each thread protects the correct chain of leaf nodes from reclamation, we take advantage of the hash flag in the bucket entries, the level tag in the leaf nodes and the knowledge that no node is inserted in a path being expanded. We use the example in Fig. 8 to better illustrate the key ideas of a safe traversal in the HHL approach. Fig. 8a shows the initial state. Assume that a thread T reached the hash level $H1$ and has updated its hazard level HL to refer to $H1$. Assume also that T was preempted in node $K1$ before reading the next-on-chain reference to $K2$. While preempted, the configuration of the chain may change due to a concurrent expansion. Later, to guarantee that when T resumes, it can safely follow the reference in $K1$, one must ensure that the reference is protected by HL. Next, we discuss three situations that can occur once T resumes from preemption.

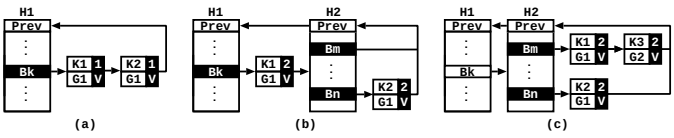


Fig. 8: Safe traversal of nodes in the HHL approach

The first situation is the case where the reference in $K1$ still refers to $K2$ as shown in Fig. 8a. Since the level tag in $K1$ is the same as HL (1=1), T can safely follow the next-on-chain reference to $K2$. The second situation is the case where the reference in $K1$ changed due to a concurrent expansion and it refers now to the hash node $H2$ as shown in Fig. 8b. Since the level tag in $K1$ is now higher than HL (2>1), T is able to detect the concurrent expansion. T then rereads the reference in the bucket entry B_k in order to understand if the expansion has already finished. As B_k is still referring to the same level, T knows that the expansion is still undergoing and, as no new nodes can be inserted during an expansion, T can safely follow the reference in $K1$ to $H2$. The last situation is the case where the reference in $K1$ also changed due to a concurrent expansion and it refers now a different node $K3$ as shown in Fig. 8c. Since the level tag in $K1$ is again higher than HL (2>1), T rereads the reference in B_k. However, in this scenario, B_k refers to the next level $H2$, thus it is not safe to follow its reference, since T can reach a node not being protected by HL. T then restarts the traversal from the reference in B_k instead of following the reference in $K1$.

In summary, when traversing a chain, T relies on the level tag to know if an expansion is happening concurrently. If T finds a level tag that is higher than the current hazard level under protection and it knows that the level in which it started the traversal has already been completely expanded, then T should not follow any reference because it can reach a node N not being protected by its hazard level.

VI. Algorithms

This section discusses in more detail the key algorithms that implement our proposal. We begin with Alg. 1 showing the pseudo-code for the *SearchKey()* procedure that given a key, returns the corresponding value associated with it.

Algorithm 1 *SearchKey(key K)*

```
1: UpdateHazardLevel(Level(ROOT_HASH_NODE))
2: UpdateHazardHash(K)
3: ⟨N, H⟩ ← SearchKeyOnHash(K, ROOT_HASH_NODE)
4: if N = Null then   // leaf node not found
5:   return Null
6: else
7:   return Value(N)
```

The algorithm starts by updating the corresponding hazard pair, using the given key K and the level of the root hash node, in order to inform the other threads that a new thread is starting a traversal procedure (lines 1–2). Next, it calls the *SearchKeyOnHash()* procedure to search for K within the hash map, starting from the root hash node (line 3). At the end, it returns the value associated with K or $Null$ if no leaf node N holding K was found (lines 4–7).

Alg. 2 then shows the pseudo-code for the *SearchKeyOnHash()* procedure given a key K and a hash node H. The *SearchKeyOnHash()* returns a tuple with two arguments – the first argument N refers to the leaf node holding K and the second argument H refers to the hash node that starts the chain where N was found (in Alg. 1, this argument is not relevant and could have been omitted). If K does not exist in the hash map, *SearchKeyOnHash()* returns $Null$ in the first argument.

The *SearchKeyOnHash()* algorithm begins by calling the *TraverseHashLevels()* procedure to traverse the path of hash levels associated with K (starting from the hash node H), until reaching the first hash node $NewH$ within that path that does not refer to another hash node (line 1). This traversal returns

Algorithm 2 *SearchKeyOnHash(key K, hash node H)*

```
1: ⟨NewH, N⟩ ← TraverseHashLevels(K, H)
2: if H ≠ NewH then   // NewH references a deeeper hash level
3:   UpdateHazardLevel(Level(NewH))
4:   return SearchKeyOnHash(K, NewH)
5: else   // H and NewH are the same
6:   HL ← GetHazardLevel()
7:   if Level(H) = HL then   // no expansion going on
8:     B ← GetHashBucket(H, K)
9:   else   // check if expansion completed ...
10:    B ← GetHashBucket(PrevHash(H), K)
11:    if EntryRef(B) = ⟨H, NextLevel⟩ then   // ... and restart
12:      UpdateHazardLevel(Level(H))
13:      return SearchKeyOnHash(K, H)
14:  while N ≠ H do
15:    ⟨NextN, LevelTag, ValFlag⟩ ← NextRef(N)
16:    if ValFlag = Valid ∧ Key(N) = K then   // leaf node found
17:      return ⟨N, H⟩
18:    if LevelTag > HL then   // check if expansion completed ...
19:      ⟨NewH, Flag⟩ ← EntryRef(B)
20:      if Flag = NextLevel then   // ... and restart
21:        UpdateHazardLevel(Level(NewH))
22:        return SearchKeyOnHash(K, NewH)
23:    if IsHashNode(NextN) ∧ NextN ≠ H then   // new expansion
24:      return SearchKeyOnHash(K, NextN)
25:    N ← NextN
26:  return ⟨Null, _⟩
```

also the reference N stored in the bucket entry within $NewH$ corresponding to K, i.e., N refers to the head of the chain of nodes where the leaf node holding K could be found.

Next, if the given hash node H is different from $NewH$, that means that at least one level was traversed by the former procedure, thus the executing thread updates its hazard level and restarts the search in $NewH$ (lines 2–4). This is necessary to synchronize the update of the hazard level with the reference N obtained from *TraverseHashLevels()*. Otherwise, H and $NewH$ are the same, thus the algorithm has the conditions to proceed with the search for K (lines 6–26). Before proceeding with the search, the executing thread T reads its current hazard level HL (line 6) and checks if there is a concurrent expansion going on (recall that a concurrent expansion can interfere with the position of a thread by placing it in a deeper hash level). If the level of H and HL are the same then, for the moment, there is no expansion going on, thus T proceeds by computing the bucket entry B for H (line 8). Otherwise, a concurrent expansion was detected, which means that T is executing in a hash node deeper than the current hazard level HL, thus T gets the bucket entry B from the previous level and checks if the expansion has been completed in the meantime, in which case it updates the hazard level to protect the hash level H and restarts the search from it (lines 11–13).

Finally, T traverses the chain of leaf nodes searching for K (lines 14–25). To keep a safe traversal, the main idea is that a next-on-chain reference is only followed if it is protected by the hazard level HL. There are three possible scenarios in this traversal: (i) K is found in a valid leaf node N and the algorithm ends returning the tuple $\langle N, H\rangle$ (lines 16–17); (ii) the full chain of leaf nodes is traversed and K is not found, and the algorithm ends returning $Null$ (line 26); or (iii) an expansion has interfered somehow with the search (lines 18–24). Two types of interference can happen: (i) T reaches a node with a *LevelTag* higher than the hazard level HL it is protecting, case in which it rereads the bucket entry B to check if the ongoing expansion has been completed in the meantime, in order to update the hazard level and restart the search as before (lines 18–22); or (ii) T reaches a new hash node, meaning that a new expansion has started, case in which T restarts the search from that node (lines 23–24).

Next, we present the procedure that supports the remove operation. Alg. 3 shows the pseudo-code for the *SearchRemoveKey()* procedure that removes a given key K from the data structure, if it exists. The algorithm also starts by updating the corresponding hazard pair and by searching for K starting from the root hash node (lines 1–3). If K is found in a leaf node N, then a three-step removal process is done (lines 5–7). First, the *MakeInvalid()* procedure marks the node as invalid (it fails if another thread has already marked the node as invalid). The *MakeUnreachable()* procedure (shown next in Alg. 4) then proceeds trying to make N unreachable. Finally, the *AddToReclamationQueue()* procedure adds N to the local reclamation queue of the executing thread.

Algorithm 3 *SearchRemoveKey(key K)*

```
1: UpdateHazardLevel(Level(ROOT_HASH_NODE))
2: UpdateHazardHash(K)
3: ⟨N, H⟩ ← SearchKeyOnHash(K, ROOT_HASH_NODE)
4: if N ≠ Null then   // leaf node found
5:   if MakeInvalid(N) then
6:     MakeUnreachable(N, H)
7:     AddToReclamationQueue(N)
8: return
```

To make a leaf node unreachable, Alg. 4 receives as arguments the leaf node N and the hash node H where N was last found. In a nutshell, the algorithm searches for the valid nodes before and after N in the chain of nodes, respectively $BeforeN$ and $AfterN$ in Alg. 4, in order to bypass node N by chaining $BeforeN$ to $AfterN$, thus making N unreachable. In more detail, the algorithm begins by calling the *SearchLeafNodeOnHash()* procedure to traverse the chain of nodes (starting from H), looking if N is still reachable (line 1). If N is already unreachable, it returns $Null$. Otherwise, it returns the hash node that starts the chain where N is found (which can be different from the initial H). While traversing the hash levels, the *SearchLeafNodeOnHash()* procedure updates the hazard level similarly to the way presented before for the *SearchKeyOnHash()* procedure.

In the continuation, if N is already unreachable, the *MakeUnreachable()* procedure simply returns (lines 2–3). Otherwise, it is ready to search for the valid nodes $BeforeN$ and $AfterN$. That process is done in three steps. First, find $AfterN$ starting from N (line 5). Second, find H starting from $AfterN$ (line 11). Third, find $BeforeN$ starting from H (line 15). If one of these three steps returns $Null$, then it means that an expansion has interfered with the process, cases in which the *MakeUnreachable()* procedure restarts, if

Algorithm 4 *MakeUnreachable(leaf node N, hash node H)*

```
1: H ← SearchLeafNodeOnHash(N, H)
2: if H = Null then   // N is already unreachable
3:   return
4: HL ← GetHazardLevel()
5: AfterN ← GetValidNodeAfter(N, H)
6: if AfterN = Null then
7:   if HL = GetHazardLevel() then   // delegation case
8:     return
9:   else   // expansion completed in the meantime
10:    return MakeUnreachable(N, H)
11: NewH ← GetNextHashNode(AfterN, H)
12: if NewH = Null then
13:   ... // same as lines 7–10
14: H ← NewH
15: ⟨BeforeN, OldRef⟩ ← GetValidNodeBefore(N, H)
16: if BeforeN = Null then   // N is already unreachable
17:   return
18: if IsLeafNode(BeforeN) then
19:   Address ← NextRef(BeforeN)
20:   NewRef ← ⟨AfterN, Level(H), Valid⟩
21: else   // bucket entry
22:   Address ← EntryRef(BeforeN)
23:   NewRef ← ⟨AfterN, SameLevel⟩
24: if CAS(Address, OldRef, NewRef) then   // try to bypass N
25:   return
26: else   // CAS failed
27:   return MakeUnreachable(N, H)
```

the interfering expansion completed in the meantime (line 10), or returns, either because the process of making N unreachable will be delegated to the interfering expansion (line 8) or because N is already unreachable (line 17). If all three steps are successful, the algorithm begins the process of trying to bypass N. A successful bypass means that a CAS operation (line 24) is successfully executed in the corresponding address of $BeforeN$. If the CAS fails, the bypass was unsuccessful and the unreachability process restarts (line 27).

VII. Experimental Results

The environment for our experiments was a machine with 2x16-Core AMD Opteron - 6274 with 32GB of main memory, running the Linux kernel 3.18.fc20 with the memory allocator jemalloc-5.0. By default, we used the LFHT data structure with a configuration of 2^4 bucket entries per hash node, a threshold of 3 for the chain node size and a threshold of 2^8 for the reclamation queue. Finally, we used a fixed size of 10^7 operations and the execution time is the average of 5 runs. To put the results in perspective, we compared the HHL method with three other approaches that we also implemented:

OF (Optimistic Free) implements an optimistic approach where each thread has a private and big enough reclamation ring buffer that fills with the nodes being removed. Each time it goes around, it reclaims the memory for the nodes in the buffer entries, before refilling them with newly removed nodes. Despite incorrect, this approach represents a best-case scenario for memory reclamation.

GPE (Grace Periods with Eras) implements a grace period method based on eras on top of our approach with the ABA problem. It uses a global clock that is atomically incremented at every removal and a local clock that every thread updates to the global clock at each quiescent state (which is declared at every operation).

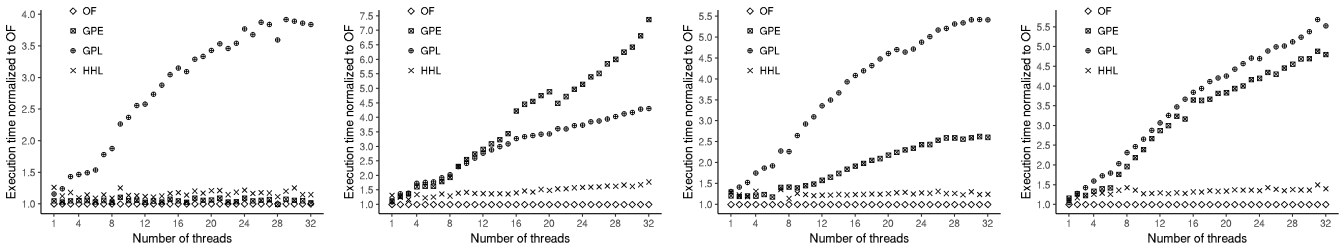
GPL (Grace Periods with Lamport clocks) implements a grace period method on top of our approach with the ABA problem, but using Lamport clocks. At a quiescent state (declared at every operation), each thread reads all of the other threads' clocks and updates its own with the maximum value read plus one.

Fig. 9 shows the execution time for the OF, GPE, GPL and HHL approaches when running four benchmarks with different ratios of insert, search and remove operations and a number of threads from 1 to 32. To better show the overhead implied by each method, all results are normalized to the OF approach.

For the benchmark with inserts only (Fig. 9a), the GPE approach behaves very closely to ideal, as the global clock is never updated, resulting in almost no synchronization for the memory reclamation done in practice. The same happens for the HHL approach, since the hazard pairs are never synchronized between threads. For the remaining benchmarks (Fig. 9b to Fig. 9d), one can observe a heavy degradation on both grace period methods, while HHL remains almost stable. This is explained by the synchronization required per quiescent state declared, which happens once per insert, search or remove operation. The reason to compare with the GPE and GPL methods was the fact that they map to the state-of-the-art Hazard Eras and Drop the Anchor methods. The Hazard Eras method follows our GPE method for clock management but, instead of doing the equivalent of a quiescent state at every operation, it does so at every node traversed in order to guarantee a memory bound. Similarly, the Drop the Anchor method follows our GPL method for clock management, but adds procedures for anchor maintenance and recovery, in order to guarantee a memory bound. As such, if any of these methods were fully implemented, they would achieve, at best, a similar performance to the one obtained with the GPE and GPL approaches. We argue that any method based on grace periods that requires at least one quiescent state per operation, would not be competitive with our HHL method in workloads that require a non trivial amount of remove operations. In Fig. 9c, we can observe that just 5% of insertions and removals is enough to more than double the execution time with 32 threads, if comparing HHL with the best grace period method.

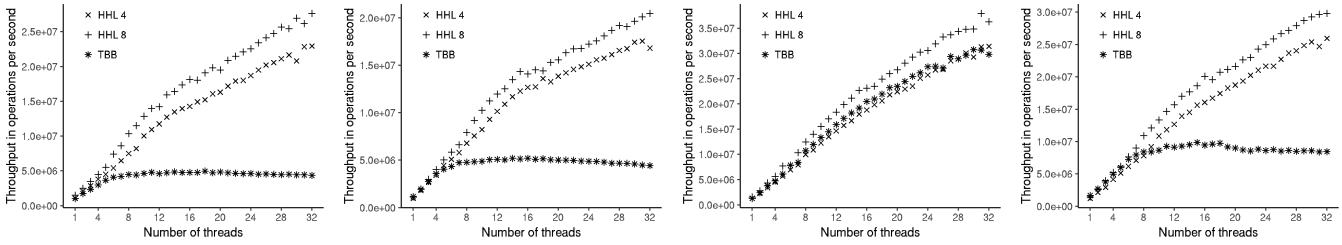
Next, Fig. 10 compares throughput (in operations per second) between the LFHT with the HHL memory reclamation method and the lock-based concurrent hash maps design from the TBB library [9]. We used the LFHT data structure with a configuration of 2^4 and 2^8 bucket entries per hash level node, which we named HHL 4 and HHL 8, respectively.

In a nutshell, both HHL 4 and HHL 8 approaches scale well in all benchmarks, while TBB shows some limitations in the benchmarks performing inserts and/or removes. For the benchmark with mostly searches (Fig. 10c), the results are very competitive but, even so, HHL 8 is still better than TBB. For the benchmarks mainly with inserts and/or



(a) inserts only (100–0–0) (b) removes only (0–0–100) (c) mostly searches (5–90–5) (d) half searches (25–50–25)

Fig. 9: Execution time normalized to the OF approach (lower is better) for the OF, GPE, GPL and HHL approaches when running four benchmarks with different ratios (*Ri–Rs–Rr*) of insert, search and remove operations



(a) inserts only (100–0–0) (b) removes only (0–0–100) (c) mostly searches (5–90–5) (d) half searches (25–50–25)

Fig. 10: Throughput in operations per second (higher is better) for the HHL and TBB approaches when running four benchmarks with different ratios (*Ri–Rs–Rr*) of insert, search and remove operations

removes, TBB suffers from a heavy performance degradation. In particular, for the benchmarks with 0% of searches (Fig. 10a and Fig. 10b), TBB does not scale when exposed to around 5×10^6 modification operations per second independently of the number of threads used, while HHL is able to scale almost linearly with any kind of operation, being able to produce about 5 times the throughput for a workload of only modification operations with 32 threads. For a 50% search ratio (Fig. 10d), the behavior is similar, but TBB stops scaling at a higher value, around 10^7 operations per second, which still corresponds to the same 5×10^6 modification operations per second. In general, these results clearly show the impact of our HHL lock-free approach compared to TBB.

VIII. Conclusions & Further Work

We have presented an efficient memory reclamation method for a lock-free hash map data structure. Our new design, named HHL (Hazard Hash and Level), uses hazard pairs to define small and well-defined regions of memory to be protected from reclamation. Since this requires very few updates to such hazard pairs during an operation, the HHL method achieves lower synchronization overheads than any of the state-of-the-art lock-free memory reclamation methods, while providing very well-defined and flexible memory bounds. Experimental results also showed that the HHL method provides a competitive and scalable thread safe hash map implementation, if compared to lock-based implementations.

As further work, we plan to study how the HHL memory reclamation method can be adapted to similar lock-free data structures and how the LFHT design can be extended to also support the removal and reclamation of hash nodes.

References

[1] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-579, 2004.

[2] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1270–1285, 2007.

[3] D. Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit, "Threadscan: Automatic and scalable memory reclamation," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 123–132.

[4] M. Herlihy, V. Luchangco, and M. Moir, "The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures," Tech. Rep., 2002.

[5] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[6] A. Braginsky, A. Kogan, and E. Petrank, "Drop the Anchor: Lightweight Memory Management for Non-blocking Data Structures," in *Symposium on Parallelism in Algorithms and Architectures*. ACM, 2013, pp. 33–42.

[7] P. Ramalhete and A. Correia, "Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation," in *Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017, pp. 367–369.

[8] M. Areias and R. Rocha, "Towards a Lock-Free, Fixed Size and Persistent Hash Map Design," in *International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2017, pp. 145–152.

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

[10] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *15th International Conference on Distributed Computing*. Springer-Verlag, 2001, pp. 300–314.

[11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs," in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2010, pp. 185–192.