On the Implementation of Memory Reclamation Methods in a Lock-Free Hash Trie Design

Pedro Moreno and Miguel Areias and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{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*, we focus on solving the problem of memory reclamation without losing the lock-freedom property. To the best of our knowledge, outside garbage collected environments, there is no current implementation of hash maps that is able to reclaim memory in a lock-free manner. To achieve this goal, we propose an approach for memory reclamation specific to Lock-Free Hash Tries that explores the characteristics of its structure in order to achieve efficient memory reclamation with low and well-defined memory bounds. We present and discuss in detail the key algorithms required to easily reproduce our implementation by others. 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.

Keywords: Memory Reclamation, Lock-Freedom, Hash Maps, Hazard Pointers

1. Introduction

Data structures are a basic programming tool that finds its way on almost every computer program or algorithm. As such, data structures are a key building block to take advantage of multiple cores efficiently and to guarantee good progress, throughput and latency properties. The traditional approach to synchronize access to critical sections in concurrent environments is to use blocking primitives, such as, spinlocks, 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 (Compare And Swap) 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 [1].

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 compatible with all lock-free data structures [2, 3, 4, 5, 6, 7], or not lock-free themselves [8, 9, 10], or depend on specific operating system or hardware implementations [11, 12].

The memory reclamation of removed elements on a lock-free data structure is not as simple as in a lock-based data structure. To ensure lock-freedom, we need to allow concurrent access 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 being removed. Over-

coming 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. To the best of our knowledge, outside garbage collected environments, there is no current implementation of hash maps that is able to reclaim memory in a lock-free manner.

In this work, we focus on extending a sophisticated implementation of a lock-free hash map data structure, named *Lock-Free Hash Tries* (*LFHT*) [13, 14], 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.

As a first approach, we started by exploring the state-of-the-art lock-free memory reclamation methods but, due to LFHT's intrinsic procedure of delegating removals under certain circumstances, all of them proved to be incompatible with the LFHT data structure as a result of an ABA problem [15]. However, in practice, we were not able to reproduce such ABA problem in our experiments, which somehow shows that the situations leading to it are extremely rare. As such, we decided to still implement some of the existing memory reclamation methods on top of this (incorrect) approach, for the sake of benchmarking and comparison of results. In any case, the design was proved to be wrong and could at any time, during execution, lead to an inconsistent state of the data structure.

As a result of this first unsuccessful attempt, we started exploring alternative designs for a memory reclamation method. This path allowed us to exploit specific characteristics of the LFHT data structure in order to be able to design a viable memory reclamation method, and also guarantee memory bounds and enhance the overall performance of the data structure. Our novel design is based on the idea of using a *hazard hash* and a *hazard level* to represent, respectively, a path and a level in the hierarchy of the LFHT data structure. This results in a small and well-defined portion of memory being protected from reclamation by each 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. As a side-effect, it presents some minor limitations. It restricts the usage of some of possible configurations of the LFHT data structure and requires the underlying memory allocator to efficiently align the memory allocation requests.

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 [16]. 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.

The remainder of the paper is organized as follows. First, we introduce relevant background regarding the state-of-the-art lock-free memory reclamation methods and regarding the LFHT data structure. Next, we present and discuss the motivation, challenges and solutions for the HHL method, with particular emphasis on the key algorithms needed to easily reproduce it by others. Then, we discuss the correctness and lock-free progress of the HHL method. At the end, we present and discuss experimental results. Finally, we present conclusions and further work directions.

2. Background

This section discusses the concepts of lock-freedom and the ABA problem threat, and the current state-of-the-art methods for memory reclamation on lock-free data structures.

2.1. Lock-Freedom & the ABA Problem

Lock-free algorithms ensure system-wide progress independently of the thread scheduling policy, i.e., whenever a thread executes some well-defined number of steps, it is guaranteed that at least one thread must have made progress during the execution of these steps. To achieve lock-freedom, we must avoid any kind of locking in

our algorithm, as a thread waiting on a lock does not make progress in any amount of time. This would not be a problem if we could control the scheduling of threads and make sure that at least one thread that is not waiting on a lock is running, but unfortunately this is not the case outside kernel space. The solution is to take advantage of atomic primitives that result in hardware specific instructions that operate atomically on memory locations.

Arguably, the most relevant atomic instruction is *Compare And Swap*, or CAS for short, which is widely supported in most modern architectures. The CAS instruction is normally used as a mean to commit a change if no concurrent task has interfered with it in the meantime. Algorithm 1 shows the pseudo-code for the CAS instruction that would be executed atomically. It receives three arguments: a memory address M; a value E, which is expected to be found in M; and a value N, to replace E in M. In a nutshell, the CAS operation atomically checks if the content in M corresponds to the expected value E (line 1) and, if so, it updates M to hold N (line 2). Otherwise, M remains unchanged. At the end, the operation returns the boolean result corresponding to whether the operation succeed or not (lines 3 and 5).

Algorithm 1 CAS(address M, value E, value N)

1: **if** $\mathit{Value}(M) = E$
2: $\quad \mathit{Value}(M) \leftarrow N$
3: $\quad$ **return** *True*
4: **else**
5: $\quad$ **return** *False*

Besides reducing the granularity of the synchronization, the CAS operation is at the heart of many lock-free data structures [17]. Usually, in a lock-free environment, we start by reading some value in a memory address, then by doing some task based on that value, and finally we try to commit the work done with a CAS instruction using the initial value read as the expected argument for the CAS. If the memory address does not contain the given expected value, the CAS simply fails, meaning that we need to redo the process as another thread has interfered with it.

However, the usage of the CAS instruction is not straightforward, as the semantic is not exactly this, because a valid expected value does not mean that it remained unchanged between the first read and the final commit. A scenario can happen where a value A is first read, then concurrent operations update the memory address from A to B and then from B to A, allowing the CAS operation to succeed with an expected value of A even though the memory's value was changed in the meantime. This is what is called an *ABA problem* [15, 18].

The prevention of the ABA problem can be done through the context of the algorithm or the use of other synchronization primitives. Preventing it through the context of the algorithm is ideal as it does not require additional hardware support. Otherwise, specific synchronization primitives, such as, double-width CAS or load-link and store-conditional primitives are required, but those are not widely available across all common architectures and practical implementations are severely limited.

2.2. Lock-Free Memory Reclamation Methods

To motivate for the problem of lock-free memory reclamation, we start by introducing Harris' lock-free linked list [19] for storing key/value pairs, which 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 the header node H. During its lifetime, a node can be in one of the following states: *valid*, *invalid*, *unreachable* or *reclaimable*. Figure 1 shows a possible configuration illustrating these states.

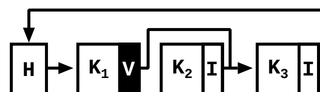


Figure 1: Node states

Node K_1 is considered valid because it is reachable from the header node H and its flag is set to valid (V). Node K_3 is considered invalid since it is reachable but its flag is set to invalid (I). Finally, node K_2 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 K_2 being unreachable, some threads

may still have local references to it, as a result of reaching K_2 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. Figure 2 shows how the remove operation changes a node state.

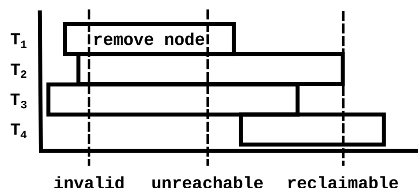


Figure 2: Node states after removal

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. 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 until the end of all operations that started before the node was made unreachable. This is the case of threads T_2 and T_3 in Fig. 2, which started their operations before T_1 has made the node unreachable. 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 T_4 in Fig. 2, which started its operation only after T_1 has made the node 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. We next present the current state-of-the-art methods for memory reclamation using these two methodologies.

Grace Periods. 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, various methods for determining the relative temporal orders between events can be used to reclaim memory with very little synchronization. Some well-know methods to establish such temporal order are the usage of eras, global epochs [8] or the Lamport clocks method [20]. All methods allow to freely control the frequency at which quiescent states are declared and at which we try to reclaim memory, but there is an associated cost regarding the amount of memory on the reclamation queue.

Although these methods do not necessarily hurt the lock-freedom property of a concurrent data structure, they do so by delegating the reclamation procedure and, as such, cannot ensure the lock-freedom of the system as a whole. The event of waiting for the declaration of a quiescent state by a single thread, renders all other threads to be unable to reclaim memory from all future nodes being removed. By itself, this does not block normal operation on the data structure, but does not allow any progress in memory reclamation, leading to unbounded memory consumption and to a possible logic lock in memory allocation.

Hazard Pointers. For space based methods, *hazard pointers* [3] 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. For the lock-free linked list example, we need two hazard pointers per thread, as we need to protect the current and the previous node that the thread is traversing. To reclaim nodes, we start by reading all hazard pointers of all threads and by comparing them against the nodes on the reclamation queue. A node not referenced by any hazard pointer can be safely reclaimed as no thread references it.

Drop the Anchor. This method can be described as a grace period based method extended with a recovery mechanism based on hazard pointers that,

despite being very expensive, is expected to run very rarely [4]. Its key idea is that when a thread T_1 identifies that another thread T_2 is not making progress in a certain amount of time, then T_2 is marked as *stuck*. A recovery procedure then tries to recover the nodes protected by T_2, i.e., the nodes between the last anchor registered by T_2 and the instant where it would update the anchor again. The recovery procedure replaces the existing nodes with new ones and marks the replaced nodes as *frozen*. This allows the remaining threads to be able to ignore the stuck threads and thus continue to reclaim memory normally. The replaced nodes remain frozen until T_2 recovers and takes care of them.

Regarding the reclamation method, apart from needing to guarantee that the removal time of a node is inferior to the clock of all running threads, similarly to grace periods, it also needs to ensure that it is greater than the clock of all recovered threads. This protects the nodes that are visible, to a thread which is either stuck or recovered, but were not frozen because they became unreachable before the recovery procedure had a chance to freeze them. The reclamation of this kind of frozen nodes can be left to the thread that caused their freeze. Performance-wise, this method approaches the grace period method but introduces a bound on the memory usage which is proportional to the number of nodes between anchors. On the other hand, it may need double-width CAS support and the cost of the recovery procedure can lead to high latency [4].

Hazard Eras. This method works similarly to grace periods, however it uses clocks not only for the removal time of every node but also for the insert time, which allows to continue reclaiming memory on the event of a thread delay or failure [5]. It uses a global clock that is increased atomically at every removal and, similarly to the way hazard pointers are updated in the hazard pointers method, when a thread reads a new reference, its local clock is updated to the global clock. The insert time of the nodes allows them to not be protected from reclamation by threads that stalled or failed before such nodes were inserted. A delayed thread can protect from reclamation the number of nodes the data structure has at the moment of the last update to its local clock. The amount of memory is bounded, but can still be high, leading to performance overheads.

Interval-Based Reclamation. This method works similarly to the Hazard Eras method, however in this method a node within a chain stores also the insertion time of the follower node, in its next reference. This information allows a traversing thread to update its local clock without consulting the global clock [6].

Comparison. All these methods have advantages and disadvantages, but it boils down to three main aspects: performance, memory usage and complexity. The grace periods method has optimal performance, as it is very simple, but the memory usage can explode rendering it to be unusable. The hazard pointers method has optimal bounds in memory, but has an extra cost of performance and it is slightly more complex to implement. Drop the anchor balances performance and memory very well, but pays the price in complexity. Hazard Eras and Interval-Based Reclamation end up balancing everything, but not excelling in any particular one. Table 1 shows in more detail the memory bounds and synchronization costs of each method.

Table 1: Comparison of the memory reclamation methods in terms of memory bounds and synchronization operations per node (T represents the number of threads, H the number of hazard pointers per thread, A the anchor interval and N the maximum number of valid nodes at any given moment)

Method	Mem Bound	Node Synch Ops
Grace Periods	*unbounded*	*none*
Hazard Pointers	$T^2 \times H$	*2 loads + 1 store*
Drop the Anchor	$T^2 \times A$	*amortized*
Hazard Eras	$T^2 \times N$	*2 loads*
Interval-Based	$T^2 \times N$	*2 loads*

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

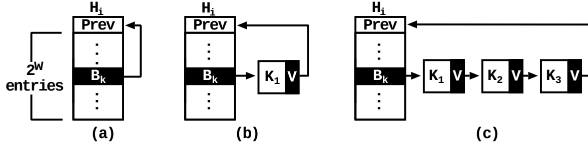


Figure 3: Insertion of nodes in a hash level

Figure 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. Figure 3b shows the configuration after the insertion of node K_1 on B_k and Fig. 3c shows the configuration after the insertion of nodes K_2 and K_3. 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. Figure 4 shows how nodes are remapped in the new level.

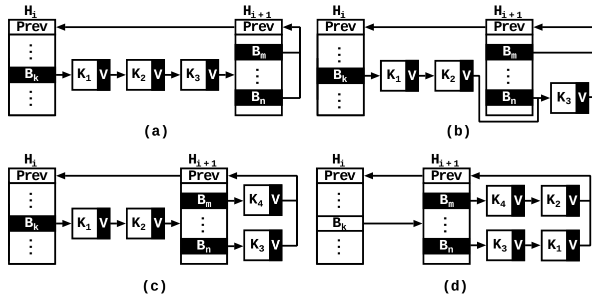


Figure 4: Expansion of nodes in a hash level

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}. Figure 4b and Fig. 4c show how node K_3 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 K_4 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.

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).

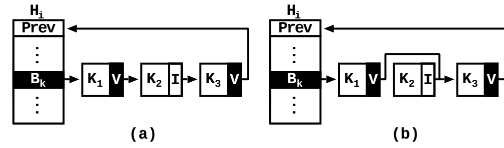


Figure 5: Removal of nodes in a hash level

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 K_1 refers to K_3.

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

pand 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*.

4. 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.

Figure 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 T_4 in Fig. 6, which started its operation after thread T_1 finished the remove operation, but before the node was made unreachable by the expanding thread T_2. In this scenario, a node can become reclaimable later than what would be expected if no delegation happened.

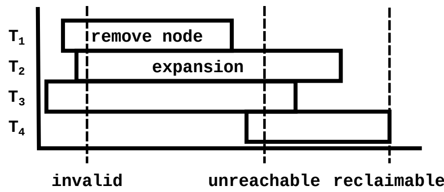


Figure 6: Node states during expansion

Avoiding this delegation mechanism is not possible since T_2 can always reinsert the node in the new hash level before realizing that it was marked as invalid and made unreachable. Figure 7 illustrates this situation in more detail. The problem resides exclusively in the case where a thread T_2, doing an expansion, reads a valid node K_3 and, before changing the corresponding bucket reference in the new level H_{i+1} in order to expand K_3 (Fig. 7a), another thread T_1 is able to invalidate K_3 (Fig. 7b) and make it unreachable (Fig. 7c). As the removing thread T_1 does not interfere with the reference in H_{i+1}, the expanding thread T_2 can succeed in updating the bucket reference B_n in H_{i+1} to K_3 and effectively reinsert K_3 making it reachable again (Fig. 7d).

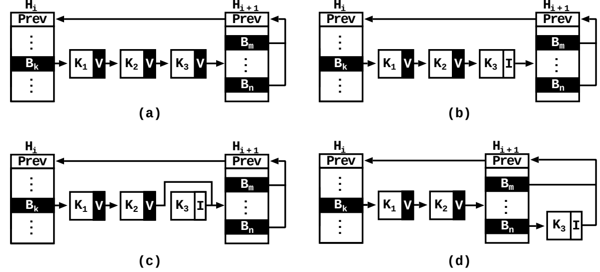


Figure 7: Reinsertion of an invalid node during expansion

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. To guarantee this, we changed the way the remove operation works 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 causes the ABA problem described earlier in Section 2. Figure 8 illustrates how the ABA problem can occur.

We consider a 1-bit tag and two expansions occurring simultaneously as shown in Fig. 8a. Thread T_1 is expanding the level H_1 (node K_1) and thread T_2 is expanding the level H_2 (node K_2) and both nodes (K_1 and K_2) are to be expanded to bucket B_n in H_3, i.e., after node K_3. Now consider that before performing the CAS to move K_2 into H_3 after K_3 (the state of K_3 becomes the first A in

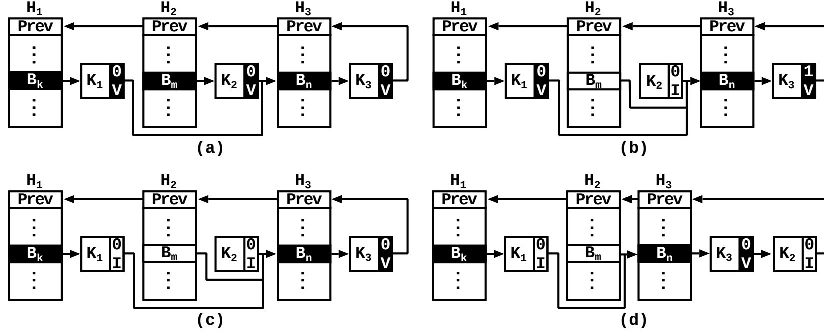


Figure 8: Reinsertion of an invalid node due to a tag overflow

ABA), another thread T_3 invalidates K_2 and, as it detects an ongoing expansion, it increments the tag of K_3 (becomes the B in ABA) and only then makes K_2 unreachable (Fig. 8b). Then, T_3 invalidates K_1 and, as it detects an ongoing expansion, it increments the tag on the expansion point, which is again K_3 (Fig. 8c). As the tag in this example only has 1 bit, it now overflows and becomes 0 again (becomes the second A in ABA). Finally, in Fig. 8d, T_2 resumes and performs the CAS on K_3 that wrongfully succeeds due to the tag overflow, thus reinserting the unreachable node K_2.

Since, in practice, this ABA situation is very unlikely to happen (we were unable to reproduced it in our experiments), we still used this approach to the delegation problem for benchmarking purposes.

5. 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. In what follows, we describe in detail the HHL approach, its guarantees and limitations.

5.1. Properties & Key Ideas

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 hash level (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

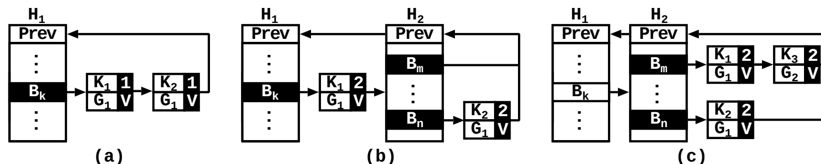


Figure 9: Safe traversal of nodes in the HHL approach

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. 9c, the value of the generation field for node K_1 is G_1, meaning that it was inserted in the hash level H_1, and the value of the atomic tuple is $<K_3, 2, V>$, meaning that it is referring to node K_3 (1st argument), it is in the hash level H_2 (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. 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. 9 to better illustrate the key ideas of a safe traversal in the HHL approach.

Figure 9a shows the initial state. Assume that a thread T reached the hash level H_1 and has updated its hazard level *HL* to refer to H_1. Assume also that T was preempted in node K_1 before reading the next-on-chain reference to K_2. 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 K_1, one must ensure that the reference is protected by *HL*. Next, we discuss three situations that can occur once T resumes from preemption.

The first situation is the case where the reference in K_1 still refers to K_2 as shown in Fig. 9a. Since the level tag in K_1 is the same as *HL* (1=1), T can safely follow the next-on-chain reference to K_2.

The second situation is the case where the reference in K_1 changed due to a concurrent expansion and it refers now to the hash node H_2 as shown in Fig. 9b. Since the level tag in K_1 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 K_1 to H_2.

The third and last situation is the case where the reference in K_1 also changed due to a concurrent expansion and it refers now a different node K_3 as shown in Fig. 9c. Since the level tag in K_1 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 H_2, 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 K_1.

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.

5.2. Delegation Problem

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 corre-

sponding 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 pre-defined 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 T_1 is not protecting N, when its hazard pair is read, and then T_1 accesses N before a second thread T_2, performing the delegation process, turns N unreachable. If the hazard pair for T_2 is read next, then it can happen that T_2 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.

5.3. Memory Bounds

Everything discussed so far is enough to make the memory reclamation method to work. However, it does not ensure a finite memory bound if an infinite number of nodes is inserted and removed from a specific chain without ever triggering an expansion. The reason is that if a thread suspends or fails in such a chain or if there is a group of threads continuously working on that chain in such a way that at any given time at least one is traversing it, this would prevent the removed nodes from ever being reclaimed. This is a rather unrealistic but nonetheless possible situation. To solve this problem, we count how many nodes each thread is protecting from reclamation, and if a thread T reaches a predefined threshold, an expansion operation is forced on the specific chain. This does not guarantee the reclamation of the previously removed nodes, but prevents T from protecting more nodes. Later, when T progresses, the previously removed nodes would be made reclaimable as no thread can acquire a hazard pair for a chain that has already been expanded.

This solution not only provides a well-defined and flexible memory bound but can also improve performance, as an expansion would likely divide the multiple threads concurrently working on a specific chain between multiple chains, thus reducing contention in that section.

The fact that we cannot force an expansion on the last hash level is not a problem. In the last level, the solution is to traverse the chain exactly as in the hazard pointers method [3], using the hazard level to inform the reclamation procedure that the hazard pointers method should be used to reclaim such nodes. Since the last level cannot be expanded, delegations cannot happen either and thus the hazard pointers method works here as intended. Note that this situation is extremely rare. For it to happen, the data structure needs to be almost full or the hash function must not be doing its job well.

5.4. Guarantees & Limitations

Guarantees. In addition to guarantee lock-freedom, the HHL method has well-defined memory bounds and low synchronization overhead. The hazard pairs and the forced expansions define a very fine control over the maximum amount of unreclaimed memory we want to allow. This amount depends on: (i) the number T of threads; (ii) the number R of remove operations per thread required to trigger the reclamation procedure; (iii) the number P of nodes protected by a single thread required to trigger an expansion; and (iv) the maximum number C of nodes in a chain. Given these parameters, the memory bound is given by Eq. 1.

$$T^2 \times (P + R + C) \quad (1)$$

The $P + R + C$ factor comes from the fact that in one reclamation procedure a thread can see $P-1$ nodes protected by a thread and in the next iteration this number could have been increased by R, as R new removals could have occurred in the same chain. Then a maximum of C nodes can still be present and then removed from the protected chain. The multiplying factor T^2 is due to the fact that for a specific thread this can happen once for every other existing thread ($T - 1$ occurrences per

thread), and every thread can be in such a situation ($T \times (T-1) \approx T^2$).

Regarding the synchronization overhead of the HHL method, in the most common case (i.e., no expansions being done), it is expected to be two atomic writes per operation, one for the hazard hash and another for the hazard level. The cases where we are traversing a chain while an expansion is occurring, we need an extra atomic read per node traversed and one atomic write to the hazard level if the expansion finishes during the traversal. Note that this is an uncommon situation, which in a lock-based approach would require a lock.

Limitations. The usage of the alignment bits to store the validity flag and the level tag limits the amount of information that we can store on the level tag, which in turn limits the maximum amount of levels that the data structure can have and consequently the minimum size of the bucket arrays. For example, assuming A as the address size in bits of the architecture in question (e.g., 32 or 64 bit size addresses), then $\frac{A}{8}$ is the number of possible addressable positions in a address sized value (e.g., in a 32 bit value we have 4 possible addressable positions). Since in our implementation a leaf node occupies 4 addresses, the number of addressable positions in a leaf node is thus $\frac{A}{2}$. Therefore, the amount of bits available for the level tag (excluding the validity flag) is:

$$log_2\left(\frac{A}{2}\right) - 1 = log_2\left(\frac{A}{4}\right)$$

This means that the maximum amount of hash levels is:

$$2^{log_2(\frac{A}{4})} = \frac{A}{4}$$

And consequently, if assuming all levels of the same size, the minimum size of a bucket array is $2^4 = 16$ entries, which seems a reasonable limit for this kind of data structure.

6. Implementation Details

This section discusses in more detail the key algorithms that implement our proposal. We begin with Alg. 2 showing the pseudo-code for the *SearchKey()* procedure that given a key, returns the corresponding value associated with it.

The algorithm starts by updating the corresponding hazard pair, with the level of the root hash node and the given key K, 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).

Algorithm 2 *SearchKey(key K)*

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

Algorithm 3 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. 2, 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.

Algorithm 3 *SearchKeyOnHash(key K, hash H)*

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

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 also returns 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 ended 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. Algorithm 4 shows the pseudocode 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. 5) then proceeds trying to make N unreachable. Finally, the *AddToReclamationQueue()* procedure adds N to the local reclamation queue of the executing thread.

Algorithm 4 *SearchRemoveKey(key K)*

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

To make a leaf node unreachable, Alg. 5 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. 5, 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

Algorithm 5 *MakeUnreachable(leaf N, hash H)*

```
1: H ← SearchLeafNodeOnHash(N, H)
2: if H = Null    // N is already unreachable
3:    return
4: HL ← GetHazardLevel()
5: AfterN ← GetValidNodeAfter(N, H)
6: if AfterN = Null
7:    if HL = GetHazardLevel()    // delegation case
8:       return
9:    else  // expansion ended in the meantime
10:      return MakeUnreachable(N, H)
11: NewH ← GetNextHashNode(AfterN, H)
12: if NewH = Null
13:    ... // same as lines 7–10
14: H ← NewH
15: ⟨BeforeN, OldRef⟩ ← GetValidNodeBefore(N, H)
16: if BeforeN = Null    // N is already unreachable
17:    return
18: if IsLeafNode(BeforeN)
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)    // try bypass N
25:    return
26: else  // CAS failed
27:    return MakeUnreachable(N, H)
```

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 the interfering expansion ended 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).

Searching for *BeforeN* and *AfterN* is an important part of Alg. 5. It is then important to understand how the actual search is done. In what follows, Alg. 6 shows the pseudo-code for finding the *AfterN* reference, given an already invalid leaf node I and a hash node H (the *GetNextHashNode()* and *GetValidNodeBefore()* procedures follow a similar pattern).

Algorithm 6 *GetValidNodeAfter(leaf I, hash H)*

```
1: HL ← GetHazardLevel()
2: if Level(H) = HL    // no expansion going on
3:    B ← GetHashBucket(H, key(I))
4: else
5:    B ← GetHashBucket(PrevHash(H), Key(I))
6: ⟨N, NodeLevel, ValFlag⟩ ← NextRef(I)
7: while IsLeafNode(N) ∧ ValFlag = Invalid
8:    ⟨NextN, LevelTag, ValFlag⟩ ← NextRef(N)
9:    if LevelTag > NodeLevel    // delegation case
10:      return Null
11:   if LevelTag > HL    // if expansion ended then ...
12:      ⟨NewH, Flag⟩ ← EntryRef(B)
13:      if Flag = NextLevel    // ... return
14:         UpdateHazardLevel(Level(NewH))
15:         return Null
16:   N ← NextN
17: return N
```

The algorithm begins by reading the current hazard level *HL* (line 1) and by getting the bucket entry B as in Alg. 3 (lines 2–5). Next, it traverses the chain of leaf nodes searching for a valid data structure (lines 7–16). There are two possible scenarios in this traversal: (i) a valid (hash or leaf) node N is found and thus returned (line 17); or (ii) an expansion has interfered somehow with the search and a *Null* value is returned to indicate that (lines 9–15). Two types of interference can happen: (i) a leaf node with a level tag higher than the level tag of I is found, case in which the process of making I unreachable is delegated; or (ii) the interfering expansion completed in the meantime, case in which we need to restart the *MakeUnreachable()* procedure from the beginning.

7. Correctness & Lock-Free Progress

In this section, we discuss the correctness of our proposal. The full proof consists of two parts: (i) prove that the proposal is *linearizable*; and (ii) prove that the *lock-freedom* property holds in all operations. We focus on the linearization proof for the algorithms described before. For that, we enumerate the linearization points, describe the set of invariants and show parts of the proof that the linearization points preserve the set of invariants.

The linearization points in the algorithms shown are:

LP_1 *SearchKey()* (Alg. 2) is linearizable at *UpdateHazardLevel()* in line 1.

LP_2 *SearchKey()* (Alg. 2) is linearizable at *UpdateHazardHash()* in line 2.

LP_3 *SearchKeyOnHash()* (Alg. 3) is linearizable at *UpdateHazardLevel()* in line 3.

LP_4 *SearchKeyOnHash()* (Alg. 3) is linearizable at *UpdateHazardLevel()* in line 12.

LP_5 *SearchKeyOnHash()* (Alg. 3) is linearizable at *UpdateHazardLevel()* in line 21.

LP_6 *SearchRemoveKey()* (Alg. 4) is linearizable at *UpdateHazardLevel()* in line 1.

LP_7 *SearchRemoveKey()* (Alg. 4) is linearizable at *UpdateHazardHash()* in line 2.

LP_8 *SearchRemoveKey()* (Alg. 4) is linearizable at *MakeInvalid()* in line 5.

LP_9 *MakeUnreachable()* (Alg. 5) is linearizable at successful CAS in line 24.

LP_{10} *GetValidNodeAfter()* (Alg. 6) is linearizable at *UpdateHazardLevel()* in line 14.

The set of invariants that must be *preserved* on every state of the data structure are:

Inv_1 For every hash level H, $PrevHash(H)$ always refers to the previous hash level.

Inv_2 A bucket entry B belonging to a hash level H must comply with the following semantics: (i) its initial reference is H; (ii) after the first update, it must refer to a node N_1; (iii) after a follower update, it must refer either to another node N_2, to the hash level H or to a second hash level H_d such that $PrevHash(H_d) = H$. If B is referring to H_d, then no more updates occur in B.

Inv_3 A node N must comply with the following semantics: (i) its initial condition is *valid*; (ii) after an update to *invalid*, it never changes to *valid* again.

Inv_4 The accessibility condition of a node N must comply with the following semantics. If N is valid then it is *reachable*. Otherwise, if N is made invalid then it must follow the sequence of stages: (i) on the 1st stage, it is *reachable*; (ii) on the 2nd stage, it moves to *unreachable*; and (iii) on the 3rd stage, it turns *reclaimable*.

Inv_5 A valid node N in a chain of nodes starting from a bucket entry B belonging to a hash level H must comply with the following semantics: (i) its initial (next-on-chain) reference is H; (ii) after an update, it must refer to another node in the chain or to a hash level; (iii) once it refers to a hash level H_d (at least one level) deeper than H, then it never refers to H again.

Inv_6 The next-on-chain reference of an invalid node N is never updated.

Inv_7 The number of valid nodes in a chain of nodes starting from a bucket entry B belonging to a hash level H is always lower or equal than a predefined threshold value $MAX_NODES \geq 1$.

Inv_8 Given a key K present in the hash map, it exists only one path P from the root hash level to the *unreachability point UP* (a hash bucket or a valid chain node), from where the node N with K should be made *unreachable*. After N is marked as invalid, the *UP* is unique.

Finally, we show the proof on how two of the linearization points, namely ***LP_8*** and ***LP_9***, preserve the set of invariants. We are not showing the proof for the remaining linearization points, but they follow a similar proof strategy.

Theorem 1. *The linearization point LP_8 preserves the set of invariants.*

Proof. Assume that a thread T is executing Alg. 4 and K exists in the hash map. Since the search operation in line 3 will necessarily find the node N that is holding K (i.e., N is not *Null*), two situations can occur: (i) T marks N as invalid; or (ii) N was already marked as invalid. For (i), Inv_3 holds – since we are updating the state from valid to invalid, Inv_4 holds – as N remains *reachable*, Inv_7 holds – because the number of valid nodes in the chain is decreased by one, and the remaining invariants are not affected. For (ii), all invariants hold, as no change in the data structure occurs. □

Theorem 2. *The linearization point LP_9 preserves the set of invariants.*

Proof. Assume that a thread T is executing Alg. 5 for a given invalid node N and that all invariants hold until the execution of the CAS operation at line 24. Previous to the execution of the CAS, Inv_4

ensures that N is reachable and Inv_7 ensures that exists only one path from the root node until the unreachability point UP of N and that UP is unique. The execution of the CAS can lead to a failure, if the value in the memory address *Address* is different from *OldRef*, or to a success otherwise. If it fails then all invariants hold, as no change in the data structure occurs. A successful CAS means that the value in *Address* changes to *NewRef*, which is a memory reference to a valid hash or leaf node. If *Address* is a bucket entry then Inv_2 holds because *NewRef* is a reference to a leaf node or to a deeper hash node that was obtained in line 5. Otherwise, if *Address* is a leaf node then Inv_3 holds – because in line 20 the third argument of the tuple maintains the leaf node as valid, Inv_5 holds – because *NewRef* is a reference to a hash or leaf node, Inv_4 holds – because the successful CAS turns N unreachable, and the remaining invariants are not affected. □

8. Experimental Results

This section presents experimental results comparing HHL with other lock-free memory reclamation methods.

8.1. Methodology

To run our experiments, we have developed a benchmarking tool that compiles the data structure and memory reclamation method together with a small module that controls the execution. Next, we describe in more detail the benchmarking tool and the environment of our experiments[1].

Figure 10 shows a visual representation of the benchmarking tool, in which the controller receives as input five parameters and communicates with the data structure through a specific interface designed to run the insert, remove and search operations. The input parameters are: (i) the number T of threads to be considered; (ii) the number N of total operations to be executed; (iii) the percentage P_i of insert operations; (iv) the percentage P_s of search operations; and (v) the percentage P_r of remove operations.

The benchmarking tool includes two stages. In the initial stage, the tool starts by launching the given number of threads T and by setting up the data structure with each thread pre-inserting the

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

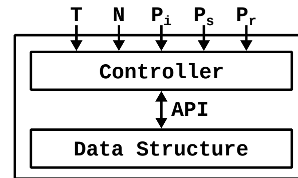


Figure 10: Benchmarking tool

keys that will be searched and removed. The total number of operations is divided equally among the T threads and each thread receives a predefined seed to be used by a Pseudo-Random Number Generator (PRNG)[2]. Then, the PRNG generates keys within a key space range, and the generated keys are distributed according to the given percentages to determine which ones correspond to insert, search or remove operations.

Figure 11 illustrates the distribution procedure for one and two threads, with seeds S_1 and S_2, respectively. However, it is important to note that the nature of the random generated values implies that the percentage of inserts, searches and removes may not be exactly the same as specified in the input parameters, but as the PRNG used has good properties and the number of operations is large enough, the actual deviation was found to be considered negligible.

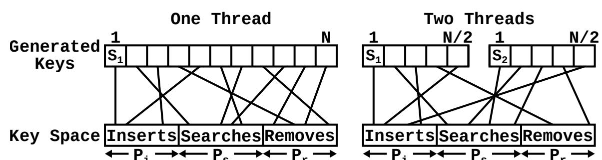


Figure 11: Distribution of keys among threads/operations

In the second stage, the tool executes the benchmark. It starts by resetting the seed of each thread to its initial value and by marking the beginning of the execution time. Then, each thread performs its set of operations according to the keys generated by the PRNG. When all threads finish their execution, the benchmark is considered complete and the execution time is presented.

In an optional third stage, we can check for the correctness of the data structure at the end, and verify if the keys in the insertion and search ranges are present in the data structure and if the keys

[2] We used the *nrand48_r()* function from the GNU C Library.

in the remove range are missing. This stage was executed for all the experiments and no errors were detected.

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 [21]. 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.

8.2. Performance Analysis

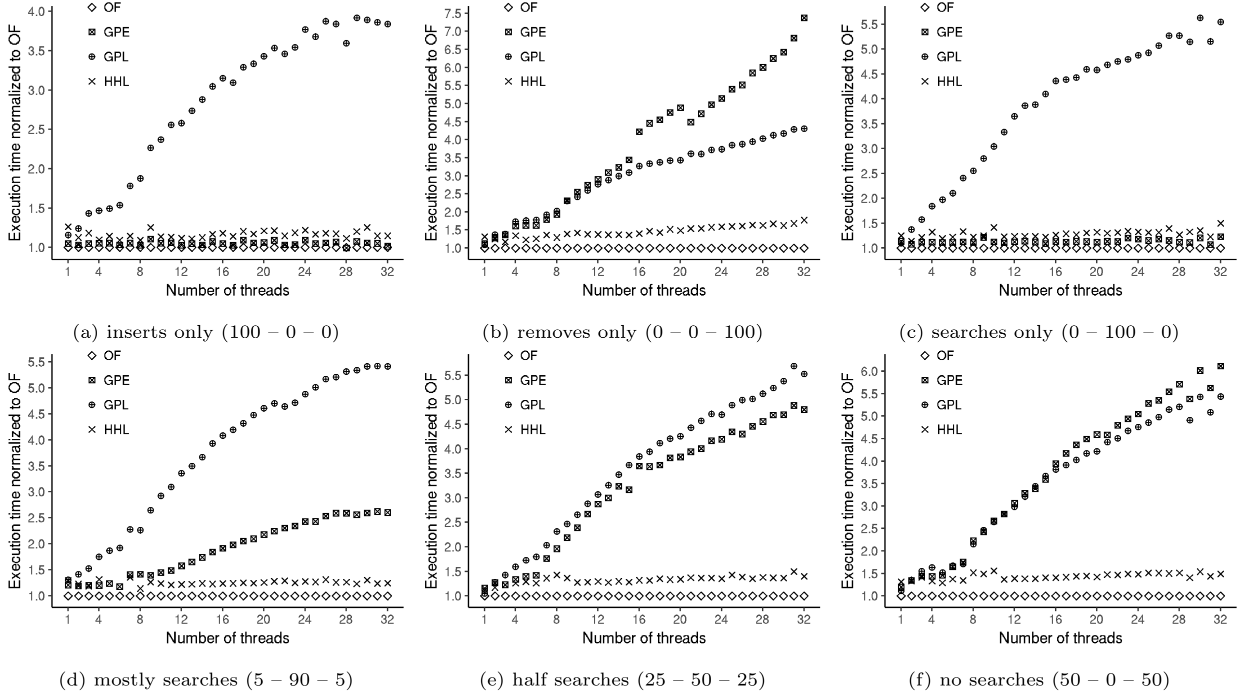
Figure 12 shows the execution time for the OF, GPE, GPL and HHL approaches when running six benchmarks with different percentages of insert, search and remove operations[3] and a number of threads from 1 to 32. To better show the overhead implied by each method, all results are normalized to the OF approach.

For the benchmarks with inserts only (Fig. 12a) and searches only (Fig. 12c), 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. 12b, Fig. 12d, Fig. 12e and Fig. 12f), 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. For example, in Fig. 12d, 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. 13 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 [16]. 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,

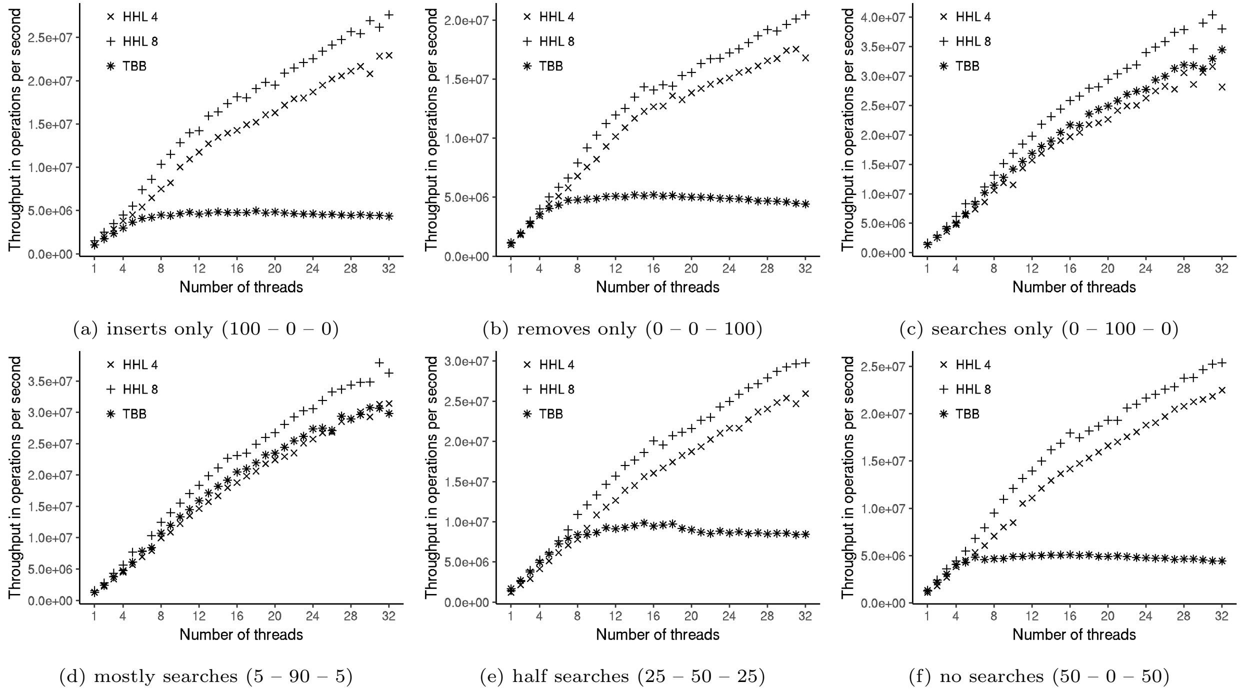
[3]We have also tested other scenarios with higher sets of keys and different mixes of the search, remove and insert operations, but have not obtained relevant results. This can be explained by the fact that all of the other scenarios can be seen as subsets of the ones that we are presenting here.



(a) inserts only (100 – 0 – 0) (b) removes only (0 – 0 – 100) (c) searches only (0 – 100 – 0)

(d) mostly searches (5 – 90 – 5) (e) half searches (25 – 50 – 25) (f) no searches (50 – 0 – 50)

Figure 12: Execution time normalized to the OF approach (lower is better) for the OF, GPE, GPL and HHL approaches when running six benchmarks with different percentages ($P_i - P_s - P_r$) of insert, search and remove operations



(a) inserts only (100 – 0 – 0) (b) removes only (0 – 0 – 100) (c) searches only (0 – 100 – 0)

(d) mostly searches (5 – 90 – 5) (e) half searches (25 – 50 – 25) (f) no searches (50 – 0 – 50)

Figure 13: Throughput in operations per second (higher is better) for the HHL and TBB approaches when running six benchmarks with different percentages ($P_i - P_s - P_r$) of insert, search and remove operations

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 benchmarks with searches only (Fig. 13c) and mostly searches (Fig. 13d), the results are very competitive but, even so, HHL 8 is still better than TBB. For the benchmarks mainly with inserts and/or removes, TBB suffers from a heavy performance degradation.

In particular, for the benchmarks with 0% of searches (Fig. 13a, Fig. 13b and Fig. 13f), 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. 13e), 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.

9. Conclusions & Further Work

We have presented an efficient memory reclamation method for a lock-free hash map data structure. To the best of our knowledge, outside garbage collected environments, there is no other implementation of hash maps that is able to reclaim memory in a lock-free manner.

Our new design, which we 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 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 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. We also intend to evaluate novel automatic memory reclamation methods, such as Automatic Optimistic Access [22] and Free Access [23], and compare their performance against our method.

Acknowledgments

This work is financed 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.

References

[1] M. Herlihy, N. Shavit, On the Nature of Progress, in: Principles of Distributed Systems, Springer, 2011, pp. 313–328.

[2] M. Herlihy, V. Luchangco, M. Moir, The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures, Tech. rep. (2002).

[3] M. M. Michael, Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, Transactions on Parallel and Distributed Systems 15 (6) (2004) 491–504.

[4] A. Braginsky, A. Kogan, 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.

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

[6] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, M. L. Scott, Interval-based memory reclamation, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2018, pp. 1–13.

[7] J. Kang, J. Jung, A marriage of pointer- and epoch-based reclamation, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 314–328.

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

[9] T. E. Hart, P. E. McKenney, A. D. Brown, J. Walpole, Performance of memory reclamation for lockless synchronization, Journal of Parallel and Distributed Computing 67 (12) (2007) 1270–1285.

[10] D. Alistarh, W. M. Leiserson, A. Matveev, 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.

[11] T. A. Brown, Reclaiming memory for lock-free data structures: There has to be a better way, in: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC '15, Association for Computing Machinery, New York, NY, USA, 2015, p. 261–270.

[12] D. Dice, M. Herlihy, A. Kogan, Fast non-intrusive memory reclamation for highly-concurrent data structures, in: Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, ISMM 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 36–45.

[13] M. Areias, R. Rocha, A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs, International Journal of Parallel Programming 44 (3) (2016) 386–406.

[14] M. Areias, 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.

[15] D. Dechev, P. Pirkelbauer, 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.

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

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

[18] D. Dechev, The ABA problem in multicore data structures with collaborating operations, in: International Conference on Collaborative Computing: Networking, Applications and Worksharing, ICST / IEEE, 2011, pp. 158–167.

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

[20] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 21 (7) (1978) 558–565.

[21] J. Evans, A scalable concurrent malloc (3) implementation for FreeBSD, in: BSDCan Conference, 2006.

[22] N. Cohen, E. Petrank, Automatic memory reclamation for lock-free data structures, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Association for Computing Machinery, New York, NY, USA, 2015, p. 260–279.

[23] N. Cohen, Every data structure deserves lock-free memory reclamation, Proceedings of the ACM on Programming Languages 2 (OOPSLA) (2018) 143.