Lock-Free Memory Reclamation for Concurrent Hash Tries

Paulo Jorge Teixeira Rosa
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência dos Computadores
2020

Orientador
Ricardo Jorge Gomes Lopes da Rocha
Professor Associado
Faculdade de Ciências da Universidade do Porto

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

Abstract

Data structures are a fundamental programming tool needed to implement programs. They are a key target to ensure concurrency properties and guarantee good performance. Our work is focused on the study of lock-free data structures, in particular the CTries (Concurrent Hash Tries) data structure, and how memory reclamation can be applied without losing the lock-free property. To the best of our knowledge, there are no implementations of CTries outside garbage collector environments. Extending the Ctries design to support lock-free memory reclamation requires adapting the data structure to achieve efficient memory reclamation with well-defined memory bounds. To achieve this goal, we study the state-of-the-art memory reclamation methods, their advantages and known problems, and how CTries can be adapted to support memory reclamation methods.
Due to the similarities between the CTries and the LFHT data structure, we focused our work on the memory reclamation method used by LFHT, named HHL (Hazard Hash Level). After extending CTries to implement the HHL method, we realized that the HHL method does not guarantee bounded memory usage in this context. To fit our goals, we then adapted HHL in order to achieve a well-defined memory bounded solution. Experimental results show that our solution introduces some overheads, mainly in the insert operation, but it is still very competitive with the current state-of-the-art methods, achieving better results than some of them.

Keywords: Memory Reclamation, Lock-Freedom, Data Structures, Concurrent Hash Tries

Acknowledgements

A giant thanks to my advisor, Prof. Ricardo Rocha, for all the support, ideas, and understanding of my problems during this work. I would like to express my enormous gratitude for this guidance in this adventure and for sharing his knowledge. I also want to give a huge thanks to Pedro Moreno for all the help and support, which provide me great learnings that were fundamental to overcome my difficulties. Finally, I would like to share a great thank you to all my friends that always support me.

Dedicated to my parents and sister

Contents

List of Tables

List of Figures

Acronyms

CTries	Concurrent Lock-Free Hash Tries
LFHT	Lock-Free Hash Tries
IBR	Interval Based Reclamation
HHL	Hazard Hash and Level
CAS	Compare and Swap
HHL BS	Hazard Hash and Level with Blocking list and Stuck array
HE	Hazard Eras
HP	Hazard Pointers
NF	No Free
OF	Optimistic Free

Chapter 1

Introduction

Computers are made of physical components and therefore follow physical laws and limitations. Back in the days, Moore's law Moore et al. [1965] defined that CPU clock speed will double every two years. However nowadays, due to physical limitations like the speed of light and the size of components, processors are practically stagnated in core speed. These limitations turned engineer's attention to the number of cores and the communications between them. As a consequence, the capability of executing multiple operations at the same time has become very important. This capability is known as parallelism.
Data structures are very important in guaranteeing some properties of programs such as concurrency properties or memory bounds. Algorithms are implemented to respect data structures' properties and definitions. Once algorithms respect data structures principles, data structures assume an important role in parallelism and can be a key point to achieve better performance.
Programs designed to run in a parallel manner require some sort of primitives to implement synchronization between the running threads or processes, since otherwise, they can end in incorrect behavior or race conditions.
Mutual-exclusion locks are probably the most used mechanism for synchronization in programs. Synchronization primitives should be used sparingly since they can introduce a non-negligible overhead on algorithms, in particular, locks can severely degrade performance as they block data access to other threads. In order to solve this problem, non-blocking approaches like : (i) lock-freedom, where is guaranteed that at least one thread make progress in a finite number of steps; (ii) wait-freedom, where every thread makes progress in a finite number of steps, and (iii) obstruction-freedom, where one thread makes progress in a finite number of steps if it is the only thread running, gained attention from the community. These approaches bring some properties to progress like progress guarantee, low latency, and scalability. In this work, algorithms are designed to execute in a lock-free manner, thus providing to our solution the guarantee that over a finite number of steps at least one operation will be concluded. To guarantee lock-freedom, data structures need to ensure progress in a non-blocking manner. Many data structures designed to accomplish lock-freedom properties rely on garbage collection environments. However, these environments usually implement memory reclamation schemes that do not guarantee lock-freedom, meaning that the memory reclamation schemes are not lock-free. This affects the lock-freedom property of the whole system because, at the moment of reclaiming memory, programs do not run in a lock-free manner.
In this work, we extend the implementation of the Concurrent Hash Tries (CTries) data structure, as originally proposed by Prokopec et al. [2012] and Prokopec [2018] to support a lock-free memory reclamation scheme. Although CTries inspired different designs, to the best of our knowledge, no version of CTries with proper memory reclamation methods exist, since all implementations are supported by a garbage collector environment.

A key problem when designing a memory reclamation scheme is to decide when a block of memory can be safely reclaimed. To be sure about such a decision, we need to guarantee that no other threads are simultaneously accessing that memory block before reclaiming it.

To understand the challenges and approaches in solving this problem, we started by studying the state-of-the-art of existing memory reclamation methods for lock-free data structures, like space and time-based methods. Some memory reclamation methods were not considered since they mostly require atomic instructions like double-width compare and swap, which are not supported by all processor architectures. Like CTries, the LFHT data structure Areias and Rocha [2016] is based on a tree-based hierarchy and has a similar traversal procedure. Due to this similarity, some ideas applied in LFHT can be applied in CTries too. In order to implement memory reclamation for the CTries data structure, we based our approach on the HHL proposal for LFHT by Moreno et al. [2019], to achieve efficient memory reclamation. Without losing lock-freedom, the HHL reclamation scheme guarantees limited bounded memory usage and scalability. However, the direct application of HHL in CTries resulted in unbounded memory usage, because HHL can not distinguish between different nodes representing the same hash levels. In order to guarantee bounded memory usage, we propose an adaption of HHL that can track the hash level collisions. Experimental results show that HHL achieves very close to optimal results and that our adaption can introduce significant overheads. Although, our solution achieves very low memory bounds, which is essential on this type of data structures.

The remainder of this document is organized as follows. First, we present some background in terms of progress guarantees, lock-free environments, and the impact of data structures in lock-freedom. Then, we go through the most relevant state-of-art memory reclamation methods and lock-free data structures. Next, we describe the implementation of the CTrie data structure and explain how it works. Then, we redesign some aspects of CTrie in order to make it compatible with memory reclamation methods, and in particular, an adaption of the HHL method is done in order to achieve a memory bounded solution. Finally, we present experimental results of different methods and compare them. In the end, final remarks and some guidelines to further work are given to inspire further studies.

Chapter 2

Background

This chapter introduces the key concepts behind our work, namely, what are progress guarantees and lock-free properties, the theoretical aspects of different lock-free data structures, and the methods used to do memory reclamation in a lock-free manner. These concepts are crucial to better understand and handle concurrent algorithms with shared memory.

2.1 Progress guarantees

Parallel programs need some kind of synchronization when accessing shared memory, otherwise, problems like race conditions can happen. In order to deal with this question, lock-based techniques have been developed. The use of lock-based primitives is relatively simple and popular, but do not give the guarantee of progress when a thread suspends, stops, or fails after acquiring a lock. In such cases, we say that the design approach is *blocking*. Alternatively to *blocking* designs, it is possible to have *non-blocking* approaches, which can guarantee progress in very different ways. Based on the differences on progress, algorithms with this properties are classified as *wait-free*, *lock-free*, *obstruction-free*, *starvation-free* and *deadlock-free*. Figure 2.1 show how progress guarantees are classified by minimal or maximal progress, dependent or independent and *blocking* or *non-blocking*, accordingly to Herlihy and Shavit [2011b].
Progress can be classified as dependent or independent. Algorithms with wait-free or lock-free properties are classified as independent since they do not depend on the OS scheduler. In these algorithms, progress is guaranteed as long there are threads scheduled to run. Other algorithms are classified as dependent, when they depend on the OS scheduler to satisfy certain properties. For example, obstruction-free algorithms require that the scheduler allow each thread to run isolated for a while.
Work done by Herlihy and Shavit [2011b] defend that shared-memory computation should rely on independent progress conditions.

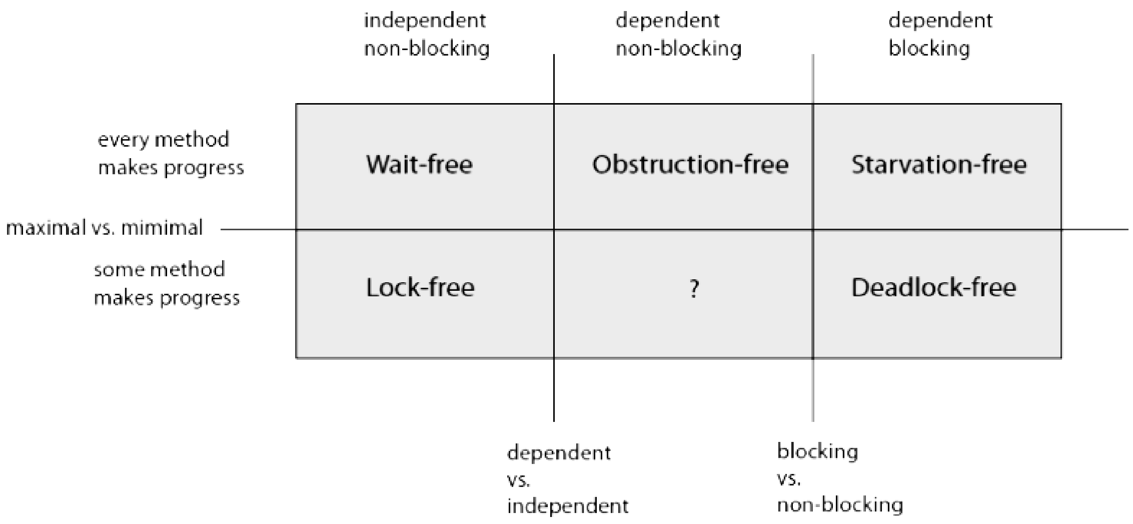

Figure 2.1: Progress guarantees

2.1.1 Obstruction-Freedom

Obstruction-free algorithms follow the idea that a thread progresses in a finite number of steps if it is the only running thread during that period of time. These algorithms do not require operations to help each other, which reduces the complexity needed to implement them if compared to the wait-free and lock-free algorithms.

2.1.2 Lock-Freedom

Lock-free algorithms guarantee system-wide progress whenever a thread executes some finite amount of steps whether by the thread itself or by some other thread in the process, i.e., when threads are run for a certain amount of time, at least one of the threads makes progress. This type of algorithm preserves the system of ending in a deadlock situation or with permanent locks, even if a thread fails. The key idea to achieve lock-freedom is to avoid any kind of locks since threads waiting on a lock cannot progress for a while.
To guarantee lock-freedom, threads can help each other in a manner that incomplete operations are able to mark their state in the shared data structures. This way, any thread can help to complete the incomplete operations. Lock-freedom is a relevant property to concurrent algorithms as it gives strong progress guarantees while not being as hard to obtain as wait-freedom due to its more relaxed nature. Lock-free algorithms are also known to have, in general, very good scalability.
The reclamation of data structures in scenarios of shared memory is a complex task, since any thread may access any valid memory location at any time.
Atomic directives working with hardware specific instructions that operate atomically on memory locations seems to be the way to follow. These atomic directives include simple atomic loads or stores or more complex instructions as atomic exchanges, bitwise operations, compare and swap directives. To our study,

the most relevant atomic directive are compare and swap directives, which is widely supported in modern architectures.

2.1.3 Wait-Freedom

A wait-free algorithm guarantees that any thread makes progress in a finite number of steps, independently of the steps executed by the other threads. Algorithms where the number of steps needed to make progress depends on the number of threads running are called *bounded wait-free*.
A data structure is considered wait-free if every operation completes after a finite number of steps, i.e., no operation will wait indefinitely to complete. Which makes it possible to calculate the worst execution time for any operation.
This type of approach also needs to ensure that threads do not end in a starvation situation. To achieve non-starvation, usually, each thread announces its current operation somewhere in memory and, when a thread makes progress, it periodically checks the announcements of other threads in order to help their operations to complete. Note that sometimes, excessive helping is not a good thing, since affect performance.

2.2 Lock-Freedom

Lock-freedom is an important property of concurrent algorithms since it brings strong progress guarantees without the complexity of wait-freedom.
An algorithm is lock-free if all the data structures where the algorithm operates are also lock-free. Otherwise, the lock-freedom property is not guaranteed somewhere in the system.
Multiple lock-free data structures have been implemented, however, most often they delegate the memory reclamation task to an independent garbage collector, which does not work in a lock-free manner. As a result, the system losses the overall lock-freedom property, because the memory reclamation procedure breaks it.
Implementing memory reclamation methods in a lock-free data structure is a complex task. In order to ensure lock-freedom on one hand, we need to allow concurrent accesses to all elements in the data structure, but on the other hand, to reclaim an element we need to guarantee that the element is not being accessed by other threads when we remove it. The delegation and determination of when reclamation methods should occur must be carefully designed.

2.2.1 ABA Problem

In our algorithms, we use CAS instructions to commit the work done by a thread. This is done in 3 steps: (i) First, we read the value we may want to change; (ii) Then we compute all the needed changes; and (iii) Finally we update the original value with the new one meanwhile computed. The CAS instruction guarantees that the computed value is only set if the original value in the data structure has not changed in the meantime. However, if the value can change multiple times and return to the original value when executing the CAS instruction, the CAS instruction will succeed, which can lead to what is known as the ABA problem.

The ABA problem can be seen as a false positive of CAS execution. For example, a thread T1 reads a value A from a shared variable V, computes all the changes, and stops. Meanwhile, a thread T2 changes the shared variable V to value B, and after that, a thread T3 changes V again to value A. Thread T1 then wakes up and executes the CAS instruction. The CAS will succeed since V have the same value A, without noticing that V has changed in the meantime. Thread T1 will act like V has not changed since the initial read and that is not true.
In some cases, although ABA occurs, the algorithm may not be affected. A classic example is the atomic add. If thread T1 reads value 10 from the shared variable V and V changes some times and backs to the value 10, the CAS instruction will give a false positive, but the result is correct.
Some primitives are immune to the ABA problem, like LoadLinked(LL), Validate(VL), and StoreConditional(SC). However, using these primitives is not the perfect solution since they are only partially supported on most architectures and because they are too strong they waste performance unnecessarily. An alternative, to solve this problem is to pack a tag with the shared variable and increment the tag when changes occur. This way, the CAS instruction will notice that the tag changed although the value is the same. Still, this is not the perfect solution since it requires the utilization of a double-CAS instruction and is limited to the tag size, which can easily overflow.
To find the ideal solution to this problem, we should look into the memory reclamation problem. It is not guaranteed that solving the memory reclamation problem prevent all cases of the ABA problem, but complete ABA solutions can be constructed by using memory reclamation solutions. In particular, garbage collection prevents the ABA problem if:

- ABA problem only lead with pointers to blocks;
- the content of a block is immutable while it is reachable by other threads;
- once a block is removed it is not reinserted before going through reclamation.

Fulfilling these requirements, the blocks follow the cycle : inserted->removed->reclaimed->allocated. If this cycle is respected, the reclamation scheme will prevent the ABA problem.

2.2.2 Persistent Pointer Problem

In some cases, lock-free algorithms require that pointers in removed blocks retain their reference values. These references are often used in traversal procedures. For example, a simple linked list may need that the pointers to the next node remain intact. Figure 2.2 shows such an example. Starting from an initial configuration with nodes N1 to Nt linked in a list (Fig. 2.2a), consider that a thread T1 stops in N2 and another thread T2 remove nodes N2, N3, and N4 from the list (Fig. 2.2b). In such a case, the nodes N2, N3, and N4 can not be reclaimed because when T1 awakes, it will reach those nodes. Reclaim them will result in a corrupted execution.

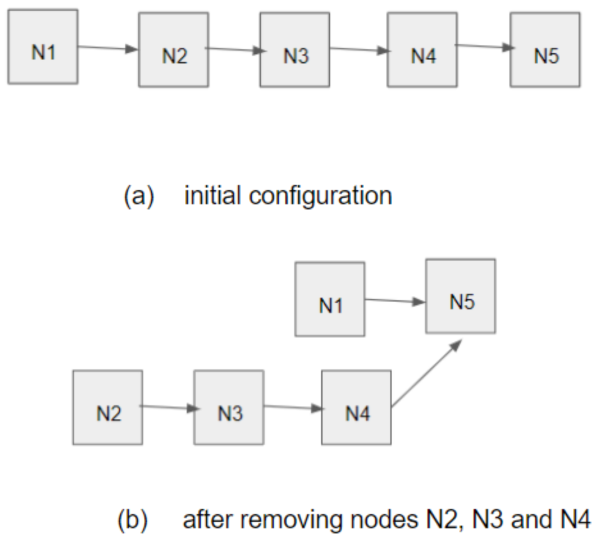
N1
N2
N3
N4
N5
(a) initial configuration
N1
N5
N2
N3
N4
(b) after removing nodes N2, N3 and N4

Figure 2.2: Persistent Pointers problem

This approach can lead to unbounded memory usage since a suspended thread can block an unpredictable number of nodes. To avoid persistent pointers, we need to redesign the algorithm such that pointers in removed blocks are immediately nullified. This approach makes the traversal procedure more difficult since we need to double-check if the previous node still points to the current one before moving on to the next one, and need to be capable of leading with null pointers.
If an algorithm has persistent pointers, it is limited to the solutions/approaches of memory reclamation that can be used. For example, hazard pointers can not be used in general, because hazard pointers allow the reclamation of blocks that are indirectly reachable from private references. And although reference counting and epoch-based solutions can be used, since they do not reclaim blocks that are indirectly reachable from a private reference, but this can lead to unbounded memory usage with persistent pointers.

2.3 Lock-Free Hash Tries (LFHT)

Lock-Free Hash tries (LFHT), as proposed by Areias and Rocha [2016], are a good starting point for our work, because of the similarities with our study, and as a fundamental background to understand the recent solution for memory reclamation presented by Moreno et al. [2019].
LFHT are a tree based data structure and has two types of nodes, *hash nodes* and *leaf nodes. Hash nodes* are used to represent the hierarchy of *hash levels*, while leaf nodes are used to store the key-value pairs. Each key is used to compute a hash, and that hash is used to map the key-value pair in the hierarchy.
Each hash node is formed by a bucket array and a reference to the previous level. All bucket entries in a *hash node* are initialized with a reference to the *hash node* itself. During execution, each bucket entry stores either a reference to a hash node or a reference to a separate chaining mechanism, that deals with the hash collisions for that entry. *Leaf nodes* also hold a reference to the next-on-chain leaf node.
To find the value hold by a certain key, we start by computing the corresponding hash. Nodes are inserted using that hash to map them in the LFHT structure, so if we follow the path given by a hash and if the

key exists, eventually we will find a leaf node containing that key and will be able to return the value hold by that node.
The insertion of a key-value pair in LFHT is illustrated in Figure 2.3.

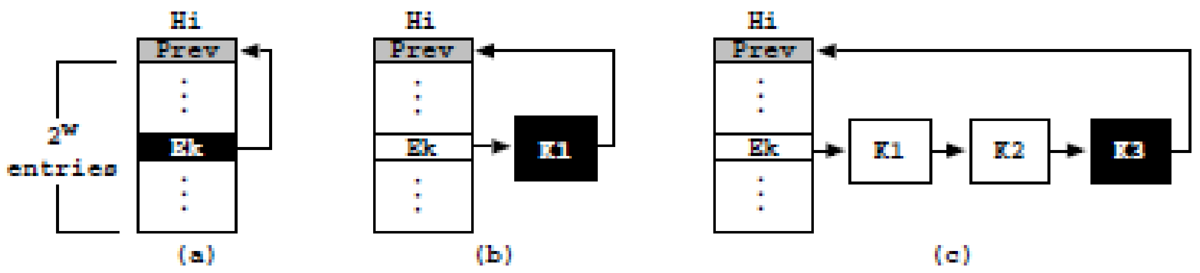

Figure 2.3: Insertion in hash level Areias and Rocha [2016]

As when searching for a node, we follow the path given by the hash of the key we are trying to insert and add it to the end of the chain hold by the corresponding bucket entry. Note that the insertion of new nodes is done at the end of the chain and any new node being inserted closes the chain by referencing back the current *hash node*.
When the number of nodes in a chain exceeds a given threshold (3 in our example), then the corresponding bucket entry is expanded with a new hash level and the nodes in the chain are remapped in the new level. Figure 2.4 illustrates the expansion operation.
When the expansion is triggered by the threshold, the current thread starts by pre-allocating a second hash level (Hi+1), with all entries referring to that level (Fig 2.4a)). Then, the Hi+1 hash level is used to implement a synchronization point with the last node on the chain that will correspond to a successful CAS trying to update Hi to Hi+1 (Fig 2.4b)). From this point, the insertion of new nodes on the bucket entry will be done starting from the new hash level (Hi+1). The remapping process will then move nodes to the correct bucket entries in the new level. And the chain of leaf nodes on a hash node (Hi) will be moved one at a time to the new level (Fig. 2.4c) to Fig. 2.4h)). In order to ensure lock-free synchronization, we need to guarantee that, at any time, all threads are able to read all the available nodes and insert new nodes without any delay from the remapping process. To guarantee both properties, the remapping process is done in reverse order, starting from the last node on the chain.

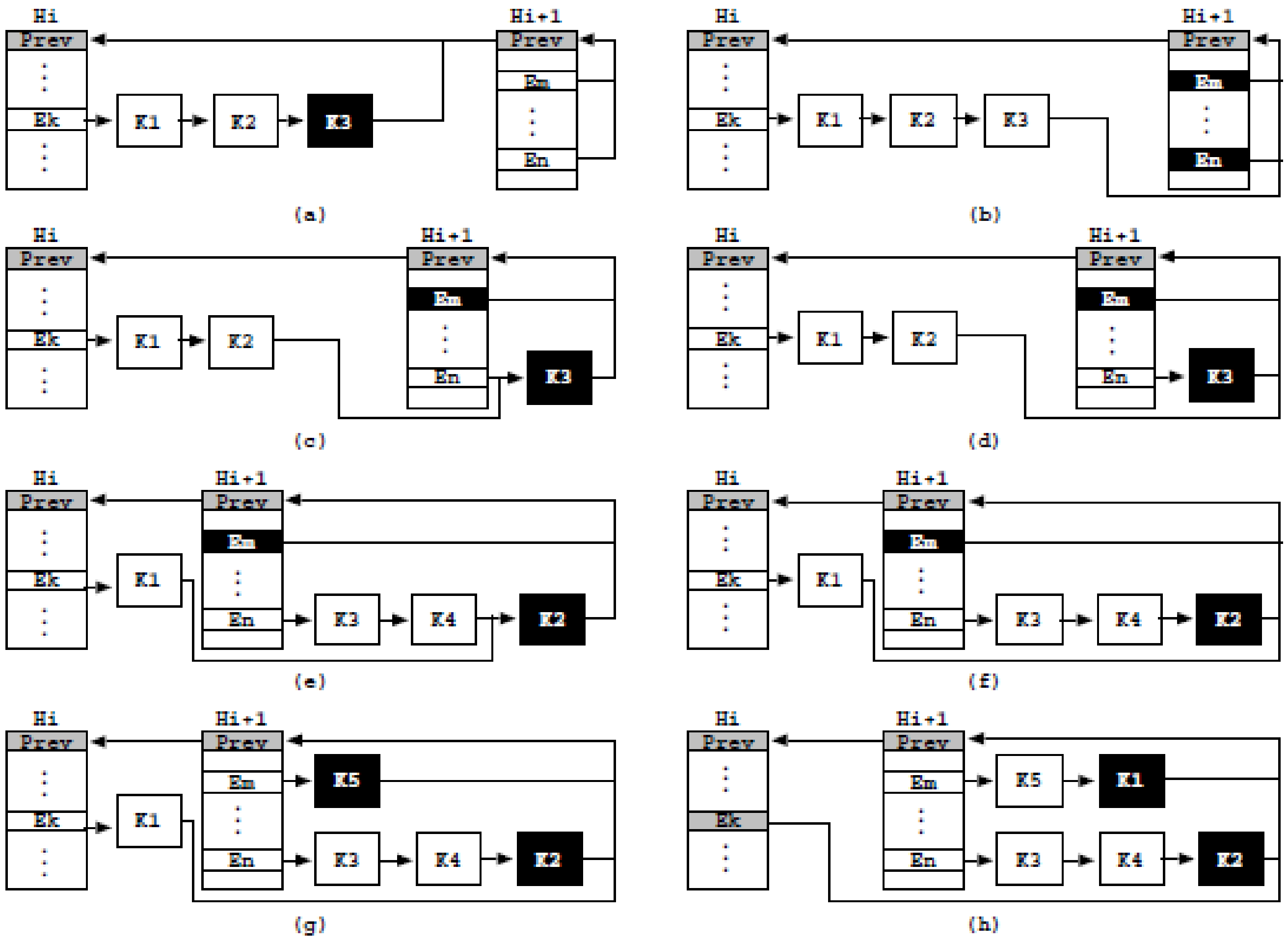

Figure 2.4: Expansion Areias and Rocha [2016]

The remove operation includes two steps: making the node invalid (as shown in Fig. 2.5(a)) and then unreachable (as shown in Fig. 2.5(b)). The invalidation step consists of finding the node to remove and change its flag from valid (V) to invalid (I). After that, to make the node unreachable, we need to, find the next valid node A on the chain and traverse the chain until finding a hash node H. If H is the same *hash node* we have started from, we continue traversing the chain until finding the last valid node before the one that we want to remove. Otherwise, if node H is not the same *hash node* we have started from, that means that an expansion is happening simultaneously and the process should restart in the next level. In the next level, we are able to make node unreachable since expansion is completed.

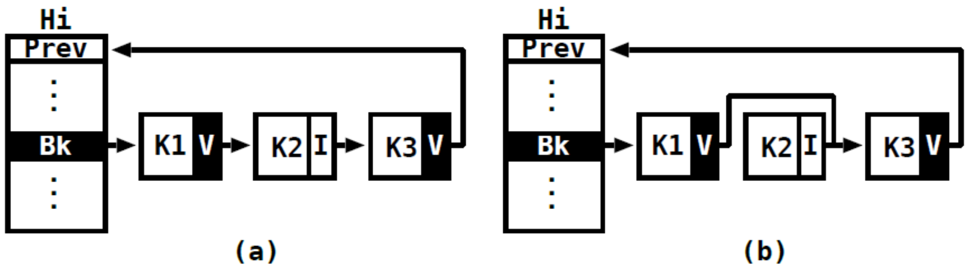

Figure 2.5: Removal in hash level Areias and Rocha [2016]

This design allows us to do updates and expansions without using the replacement of data structures, avoiding the need for memory recovery mechanisms.

2.4 Memory Reclamation

As we already mentioned, to truly guarantee lock-freedom, we need to guarantee also the lock-freedom of the memory reclamation scheme since, in the end, the data structure algorithms and the memory reclamation algorithms should be combined in order to make progress like expected. If that is not taken into consideration, our program might consume unbounded memory and lead to a logic lock when requesting more memory, which is not available. The CTries data structure Prokopec [2018] was initially defined for an environment where a garbage collection is provided, thus making it unusable in environments without such support.
The design of a memory reclamation scheme may assume that fair scheduling will not happen and that operations should progress independently of scheduling. To provide lock-freedom properties on memory reclamation scheme, at least one thread needs to be able to make progress in a certain number of steps but, on other hand, lock-freedom permits the starvation of a thread.
When we think about the memory reclamation scheme, we need to be aware that a thread can be suspended for an undetermined amount of time in a node. While this thread is suspended, that node can not be reclaimed. This can be a problem if we want to guarantee a memory bound since, eventually, a program that allocates memory and does not frees it will consume unbounded memory.
Memory reclamation schemes should guarantee that systems have available memory to complete any operations while running, otherwise will violate the lock-free property. Usually, memory reclamation requires information from all running threads, which can affect the scalability that lock-free algorithms usually have.
Most often, data structures are not designed to include memory reclamation methods. Many data structures are implemented without taking memory reclamation into consideration and need to be modified afterward in order to combine with memory reclamation methods.
Next, we will explain the most interesting memory reclamation methods.

2.4.1 Optimistic Reclamation

Optimistic reclamation is based on the low probability of a thread be accessing a memory block that was removed some time ago.
This approach is theoretical wrong since a thread can continue referring to that memory block. However, since this method is quite simple and requires minimum synchronization and overhead, can be applied in practice, as a baseline to compare with other methods.

2.4.2 Grace Period

When a thread accesses shared resources we say that the algorithm is running in a critical section. Critical sections guarantee that no other threads are interacting with the same shared resource simultaneously.
This introduces two other important notions, the notion of quiescent state and grace period.
When a thread runs outside of any critical section, we say that the thread enters a quiescent state. This state guarantees that this thread has no access to shared resources, and this, cannot cause problems to the other threads.
A grace period is a period of time where all threads have been in one or more quiescent states. For example, consider the case where a node is made unreachable. We know that, when a grace period happens, the node can be safely reclaimed. The fact that a grace period as passed guarantees that no other thread is referring to the unreachable node.
Based on this idea, several techniques, which have the objective of determining the temporal order between events, can be used to efficiently implement memory reclamation.

2.4.2.1 Lamport Clocks

Lamport Clocks is one of the most popular techniques to determine grace periods. In this scheme, all threads have an internal clock and all threads can read the internal clocks but only the owner thread can update its internal clock. This internal clock is the key to threads synchronization. When a thread wants to mark an event, it starts by reading all clocks and by updating its clock to have the maximum value (max value of all threads + 1).
This way, it is easy to determine which events occur before and after a certain event. If event A is marked with lower values than event B, we know that event A happens early than event B.
Consider the event that makes node A unreachable, and after that, we try to reclaim the memory addressing node A. If all threads update their internal clocks when they enter a quiescent state, to know if a grace period has passed, we only need to read all thread's internal clocks. If all threads have updated their clocks to a value indicating a quiescent state after we made node A unreachable, a grace period has been elapsed and it is safe to reclaim the memory of node A.
This approach can reduce the scalability of programs and lead to more memory consumption since its cost is proportional to the number of threads.

2.4.2.2 Global Epochs

Global epochs are another way of determining the temporal order between events, which uses a shared clock that can be updated by all threads. This clock serves the purpose of marking quiescent states and events and, for that, threads use atomic increments. Since it relies on a shared variable to implement the

clock, this can degrade the performance of the system.
Moreover, this method does not guarantee progress in the process of memory reclamation, which can lead to problems with unbounded memory consumption and logic locks.

2.4.3 Hazard Pointers

The Hazard pointers Michael [2004] scheme is one of the most popular lock-free memory reclamation schemes. Hazard pointers rely on the idea of collecting all memory references that threads are currently accessing, in a way that such information is sufficient to guarantee that we do not reclaim any memory currently used by other threads. In practice, hazard pointers are shared variables that store pointers to the nodes that are currently been used by a thread.
When removing a node, this scheme first marks the node as invalid and then makes it unreachable. Unreachable nodes are then collected into a local reclamation queue. Nodes will wait in the reclamation queue until the reclamation procedure starts. When the reclamation procedure executes, all nodes in the reclamation queue are compared with the references stored in the hazard pointers, and if a node is not referred by hazard pointers, then no threads are currently accessing him, which means that it can be safely reclaimed.
Usually, the reclamation procedure is executed when a certain threshold in the reclamation queue is reached. This permits to adapt the threshold in a way to adjust memory consumption and performance.
We need to be careful about the compatibility of this method with our data structure since it is possible that hazard pointers can not be applied to some data structures. As an example, consider the Harris-Herlihy-Shavit linked list Herlihy and Shavit [2011a].
Assume a linked list formed by 4 nodes, connected by numeric order, and a thread searching for key 4, that is suspended on node 2. While sleeping on node 2, other threads remove nodes 2 and 3 and disconnect them from the linked list. At this moment, both nodes 2 and 3 are referred to in the reclamation queue. Since one thread is sleeping in node 2, this node is guarded by its hazard pointers, but node 3 is free to be reclaimed. If node 3 is reclaimed, and later the thread on node 2 wakes up, an error will occur because the thread will try to achieve the next node referred to in node 2, that is node 3, which does not exist anymore.

2.4.4 Drop the Anchor

The Drop the Anchor method is a combination and improvement of grace period methods and hazard pointers. Drop the Anchor extends the grace period method with a field to mark the removal time, frozen nodes and store an anchor and the thread state into the thread internal clocks. Two bits of internal clock pointer are used to mark if a thread is idle, running, stuck or recovered. This allows us to identify the threads that are in an idle state, and thus ignore that threads clocks during the reclamation process. It also allows to, mark a thread as stuck if the thread does not make progress for a while, a case in which it should then call a recovery procedure. The anchor is basically a hazard pointer, updated when certain nodes are accessed.
The recovery procedure has the task of recovering nodes that can not be reclaimed because a stuck thread is on it. The recovery scheme replaces the existing nodes with new ones and marks the replaced nodes as frozen. Now, the remaining threads can ignore the stuck threads and continue to reclaim memory. Replaced nodes continue frozen until the stuck thread recovers and deal with them.
Then, recovery starts at the anchor referenced by the stuck thread and marks all nodes as frozen, copying

valid nodes to a new list until the next anchor point. After that, the node previous to the stuck thread's anchor reconnects to the new list, the thread's clock is updated and the thread is remarked as recovered. To maintain the lock-freedom property, all threads including the stuck ones should be able to assist in these procedures.
With this method, it is possible to achieve similar to the performance grace period, and at the same time have bounded memory usage. However, the recovery procedure can degrade performance and that is why it is so important to decide how and when the recovery procedure should be used.

2.4.5 Hazard Eras

Hazard eras are based on grace periods too, but extend the approach by using a new clock to store the insertion times. This improvement makes it possible to continue reclaiming memory when a thread delays or fails.
This method has a global clock that is updated at every remove operation and, in the same way, that hazard pointers are updated, when a thread reads a new reference, it updates its local clock value to the global one.
With the extension of the insertion time, we are now able to ignore stuck threads when trying to reclaim nodes that are created after the stuck thread. Since it is guaranteed that if the thread remains stuck, it will not be able to access the nodes created after the stuck time.
Hazard eras provide bounded memory usage but imply great memory overhead and performance degradation.

2.4.6 Interval Based Reclamation

Interval Based Reclamation (IBR) is based on the Hazard Eras method. This method proposed by Wen et al. [2018] uses a global epoch counter to mark the time. Each block holds a birth epoch, stored when it is created, and a retire epoch, that marks the epoch when removed. Each thread has a local retirement list to store retired blocks that will pass through the reclamation scheme. This way, by identifying a finite range of epochs, the reclamation scheme can detect the blocks and threads whose lifetime's epochs are intersected. This guarantees bounded memory usage since the number of blocks between the two epochs is finite. IBR has some versions according to the data structure being used.
In the case of persistent data structures, where all intermediate pointers are immutable, modifications on the data structure rely on linking new nodes to unchanged parts of the structure. IBR operations resemble Epoch based Reclamation, except that instead of reserve all unretired blocks before a given epoch, reserve only blocks whose lifetimes intersect reserved epochs. Threads reserve the epoch in which it first reads the root. Read the global epoch and root reference, posts the epoch to a global "tracker" (global clock) to reserve it, executes a write-read fence, and then double-checks the global epoch to assure it has not changed. This mechanism guarantees that the root's content was active during the reserved epoch. Once persistent data structures have immutable pointers, blocks reachable from the root are also active during reserved epochs. This way, threads reserve all blocks that could possibly read during the execution of the current operation.

2.4.7 HHL

Hazard hashes and levels is the solution proposed by Moreno et al. [2019], to implement memory reclamation in the LFHT data structure. This design is based on the idea of using hazard hashes to define paths in the hash hierarchy and hazard levels to protect the level in that hierarchy. Since we can use a hash-level pair to represent a specific chain of nodes, a hazard pair (hazard hash and hazard level) is enough to protect a group of nodes .
This is a great advantage comparing with other methods that can only protect a single node per hazard pointer. This strategy reduces the synchronization overhead without letting memory bounds explode.
To implement this solution, the original LFHT data structure suffered some changes, like adding a hash flag on bucket entries, and a generation field and a level tag in leaf nodes. With this new scheme, all threads help to finish all ongoing expansions in a path before inserting new nodes in such a path, ensuring that only one expansion will occur at the same time in a path.
To prevent single threads from blocking an unlimited number of nodes from being reclaimed, if the number of nodes blocked from reclamation by a hazard pair exceeds a certain threshold, an expansion is forced on that chain.
The list of hazard pairs need to be read twice when performing the reclamation procedure, as a node can be added to a reclamation list before becoming unreachable due to the delegation process.
With this scheme, we have enough information to hold the chains where the node has been during his lifetime. The reclamation procedure starts by reading twice the list of hazard pairs from all the running threads, and stores a copy of each one. If a node is not in any of both copies of the list of hazard pairs, it is safe to reclaim it.
Since this method can count how many nodes each thread is blocking, it guarantees bounded memory consumption, since just a small gap of memory will be blocked from the reclamation process. Following this approach, the number of updates required on hazard hashes, and levels remain small as well as the synchronization overhead.

2.4.8 Memory Reclamation Conclusion

Other solutions exist in the literature, such as, the limbo lists by Kung and Lehman [1980] and the free access (general lock-free memory-reclamation) scheme proposed by Cohen [2018].
Independently of the solution, in the end, performance, memory usage, and complexity still are the key aspects when designing a memory reclamation scheme and we need to focus on the ones that are compatible with our data structure and that can run inside our system limits.

Chapter 3

Concurrent Lock-Free Hash Tries (CTries)

In this chapter, we present our implementation of the concurrent lock-free hash tries (CTries) data structure, and, for that, we follow the proposal by Prokopec et al. [2012] and Prokopec [2018]. The CTries map key-value pairs into a tree-based hierarchy. Like in the LFHT data structure, the CTries use the hash calculated from the key to finding the path leading to the node that holds the corresponding key. The CTries can include six different types of nodes, as represented in Fig. 3.1 and in Listing 3.1:

- ANodes are used to implement the tree hierarchy. They start as a 4 size array (narrow) and can be expanded later to a 16 size array (wide). They can hold pointers to any kind of nodes;
- SNodes are the leaf nodes. They guard a key-value pair, the corresponding hash, and a pointer to save the state of pending operations;
- FNodes are used to mark freezing nodes and they hold a pointer to the frozen node;
- ENodes are expansion nodes and are used to handle the expansion operation;
- CNodes are compact nodes, and are used to mark a compact procedure;

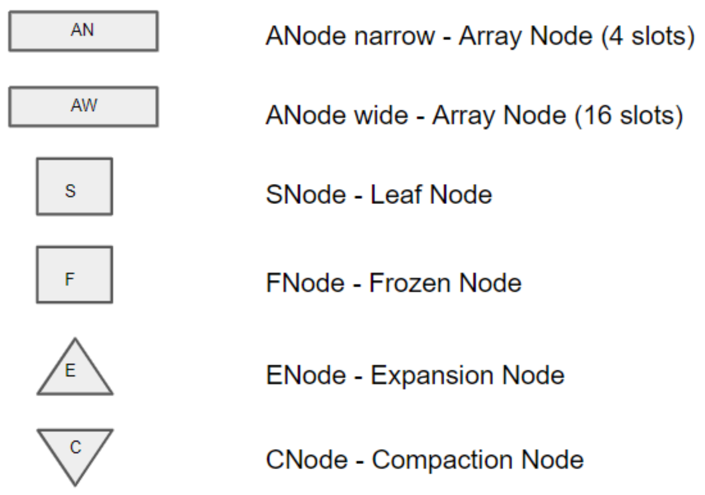

Figure 3.1: The Ctries type of nodes

The CTries data structure is identified by a root reference. Initially, the root reference points to an empty ANode of size 16 (wide). Figure 3.2 shows an example of the data structure organization in levels. In what follows, we describe the behaviour of the data structure for the six basic operations: lookup, insert, remove, expansion, compaction and freeze.

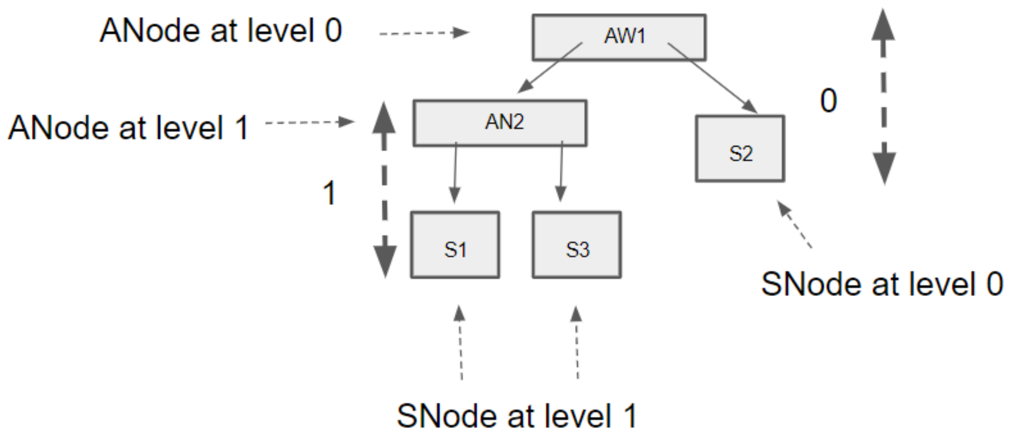

Figure 3.2: Basic CTrie structure in levels

In the CTries each ANode represents a level and each time an ANode has traversed the level increment,

```
typedef struct {
    void **array;
    int size;                          // 4 or 16 entries (narrow or wide ANode)
}*ANode

typedef struct {
    size_t hash;
    size_t key;
    size_t val;
    void *pending;    // used to annouce pending changes, also used to mark the
        node as frozen
}*SNode;

typedef struct {
    ANode cur;                                               // always NULL
    ANode frozen;                                        // the frozen ANode
    ANode prev;                                        // the previous ANode
    int prev_pos;                       // entry of FNode in the previous ANode
    int level;                                              // level of FNode
}*FNode

typedef struct {
    ANode narrow;                      // the narrow ANode that will be expanded
    ANode wide;                    // the wide ANode that replace the narrow one
    ANode prev;                                // the previous traversed ANode
    int prev_pos;                  // the entry of ENode in the previous ANode
    int level;                                          // the level of ENode
}*ENode

typedef struct {
    ANode cur;                                           // always not NULL
    ANode prev;                                // the previous traversed ANode
    int prev_pos;                   // the entry of CNode in the previous ANode
    int level;                                              // level of CNode
}*CNode

#define IS_SNode(entry)  (GET_TAG(entry) == SNODE_TAG)
#define IS_ANode(entry)  (GET_TAG(entry) == ANODE_TAG)
#define IS_ENode(entry)  (GET_TAG(entry) == ENODE_TAG)
#define IS_CNode(entry)  (GET_TAG(entry) == CFNODE_TAG &&
                          ((CNode)UNTAG(entry))->cur != NULL)
#define IS_FNode(entry)  (GET_TAG(entry) == CFNODE_TAG &&
                          ((CNode)UNTAG(entry))->cur == NULL)
```

Listing 3.1: The fields defining the six different types of nodes.

Fig. 3.3 demonstrates the level delimitation.

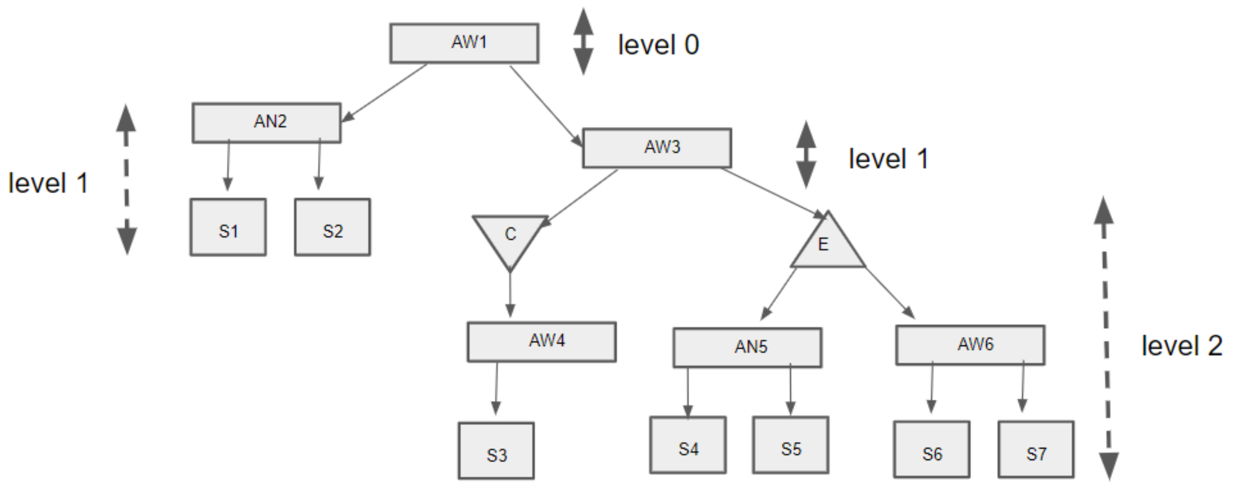

Figure 3.3: The Level delimitation in the CTries

3.1 Algorithms

3.1.1 Lookup

The goal of the lookup algorithm is to find the node associated with a given key and return the associated value. For that, it needs to be able to follow the path leading to the wanted node. Listing 3.2 shows the pseudo-code for the lookup_key() procedure that implements the lookup algorithm. The procedure receives the key it is searching for, the hash associated with the key, and the current level and ANode in the CTrie structure. In case of success, it returns the corresponding SNode for the given key or NULL in case of failure. The procedure starts by calculating the position in the ANode array (line 3). The correct path to follow is obtained by reading the corresponding 4 bits of the hash at each level. This searching mechanism is the same used by all procedures acting on the data structure. For example, consider that thread T1 is searching for k0, and that the calculated position for the root level is 0, so, we start by reading the first entry on the root's array. If the entry is empty (NULL), so the key we are searching for is not assigned in the CTrie and the algorithm returns NULL (lines 6-7), representing key not found. Assume that instead, the lookup algorithm encounters an array node (ANode). The procedure continues following the respective ANode referenced on the corresponding array entry (lines 13-14). Next, in the second level, the algorithm finds a leaf node entry (SNode), then it proceeds in order to confirm that this node has the same key we are looking for (k0). If so, it returns the node (lines 8-11). Otherwise, if the key is not the same we can assume that the key we are looking for it is not inserted in the data structure and return NULL (line 12). If lookup encounters one of the special nodes, expansion nodes (ENode), frozen nodes (FNode), or compaction nodes (CNode), the algorithm proceeds the search using the old view of data structure, where does not exist pending operations. Finding an ENode, lookup proceeds into the stable version with old narrow ANode, that is frozen or will be soon (lines 15-16). If not frozen yet, the lookup continues normally until finding an empty entry or a SNode. If frozen, the frozen node permits the lookup to view the old information, information that is being replicated into the new wide ANode (procedure explained forward). Finding an FNode, we proceed into frozen ANode (lines 17-18). Finally, if lookup encounters a CNode, representing a Compact node, it follows the same principle as for the ENodes, looking into the old stable version of nodes (lines 19-20).

```
 1 SNode lookup_key(size_t key, size_t hash, int level, ANode cur){
 2    ANode next;
 3    int pos = GET_POS(hash, level, cur->size);
 4    void *entry = cur->array[pos];
 5    void *node = UNTAG(entry);
 6    if (node == NULL)                                   // key not found
 7       return NULL;
 8    if (IS_SNode(entry)) {                    // if same key, return value
 9       SNode snode = (SNode)node;
10       if (snode->key == key)
11          return snode;
12       return NULL;
13    } else if (IS_ANode(entry))                       // continue search
14       next = (ANode)node;
15    else if (IS_ENode(entry))                        // use old version
16       next = ((ENode)node)->narrow;
17    else if (IS_FNode(entry))                       // go inside frozen
18       next = ((FNode)node)->frozen;
19    else if (IS_CNode(entry))                       // go inside frozen
20       next = ((CNode)node)->frozen;
21    return lookup_key(key, hash, level+1, next);
22 }
```

Listing 3.2: Pseudo-code for the lookup_key() procedure

We assume that lookup does not need to worry about nodes that are close to being inserted. For example, if thread T1 starts searching for k1 and meanwhile thread T2 executes an expansion on ANode A1 that holds k1, T1 will catch the ENode entry and continue the search using the narrow ANode A1. Then, thread T2 finishes the expansion and removes k1 after that. T1 awakes when T2 finishes, and because T1 is referring to the ANode A1, the node with k1 will be found by T1 although it was removed by T2, but T2 has removed k1 on wide ANode A2 and not in A1. This is not a problem because, at the moment that T1 reached A1, the key was assigned to the structure. So in a way, it existed at the moment we search, but not when we return the value.
Following this implementation, lookup does not help pending operations to complete but guarantees the lock-free property.

3.1.2 Insert

The Insert algorithm searches for an ANode, where a new SNode will be inserted, or for a SNode that will be replaced. Different from lookup, the insert cares about ongoing changes and helps to complete them. Listing 3.3 shows the pseudo-code for the insert_key() procedure that implements the insert algorithm. This procedure receives the pair key-value to be inserted and respective hash, the level it is traversing, the corresponding ANode of that level, and the previous traversed ANode. It returns success if the insert succeeds. In case of procedure failure, the algorithm restarts again until obtains a successful result.
The insert algorithm follows the same steps to calculate the position of ANode's entry as in the lookup procedure (line 2). The Search mechanism is followed until it finds an empty entry or nodes different of ANode type. In the case of reaching an empty array entry, the SNode with the corresponding key-value pair is added to the data structure on that entry (lines 5-6). To do that, it calls the insert_snode() procedure represented in Listing 3.4. Procedure insert_snode() creates on the thread's local memory a new SNode with the given key-value pair (line 2) and uses the CAS directive to atomically update the data structure if the corresponding array entry is still empty (line 4). If other updates have occurred on that entry and the CAS directive fails (lines 6-7), then the insert_key() algorithm will start again. When insert_key() algorithm ends up on a SNode entry (lines 11-12), we need to consider different possibilities, as shown in collision_snode(), represented in Listing 3.5.
First of all, we need to check if the node is not in a frozen state, checking the pending field of SNode (line 4). Assuming that SNode is not frozen if the algorithm finds another SNode holding the same key that it is trying to insert, a replacing situation will happen, and a new SNode is created on the thread's local memory (line 6). After that, the SNode's pending field is updated to refer the new SNode created (line 8). At this point, the new SNode is assigned to be inserted but lookup still does not see it. The effective change on the structure happens when the entry is replaced by the new SNode (line 9). Every thread that finds a reference on a SNode's pending field should proceed in order to help to complete the operation. For that, threads update the reference hold by previous array entry to the reference hold by SNode's pending field. If for some reason, a thread fails a CAS on SNode's pending field, it reenters on the previous entry (line 13).
Otherwise, if the array entry reached by the algorithm is occupied by a SNode with a different key, then a collision is triggered. To treat collisions, the algorithm takes in consideration if the current ANode is wide or narrow. If the array is fully expanded (wide), then we need to insert a new level at the structure in order to solve the collision. To do that, we create a new narrow ANode to represent the next level of the data structure. To create a new level, threads start by creating a new narrow ANode (line 17) and use the same mechanism as used for the replacement (lines 18-23). i.e., referring the new ANode in the SNode's pending field, thus providing sufficient information to the other threads be able to help to complete the operation. The new Anode is created together with a copy of the old SNode in the corresponding position calculated by the hash. The algorithm then fails and reenter from the root reference since now the data structure has one more level on that path, this usually solves the collision. However, this is not the only collision scenario. Another collision scenario is when the algorithm finds an array that is not fully expanded, with 4 slots only (lines 14-15). Differently from the previous case, this time, the collision can be solved without the need of creating new levels in the structure. So, we expand the size of the array to 16 slots, reducing by 75% the probability of collision. For that, the algorithm calls the expansion() procedure. This procedure is responsible for the expansion operation and provides the information needed to other threads to be able to help to complete the procedure. When the expansion is completed, the data structure will hold the new wide array and we try the insertion again. Consider now that, the thread finds a frozen node (identified by the pending field), then the thread restart from root reference (line 30). This process of reentering from the beginning of the "hierarchy" guarantees that the algorithm reaches

nodes with the necessary information to help the pending changes started by other threads. Following this protocol, when the algorithm finds pending changes on pending fields, it completes them first in order to submit the changes to the data structure as soon as possible (lines 26-28).
If insert_key() algorithm reaches an ENode or a CNode entry, completes the pending procedure first, reenters from the root reference, and then performs insertion (lines 13-17).

```
 1 int insert_key(size_t key, size_t val, size_t hash, int level, ANode cur, ANode prev) {
 2    int pos = GET_POS(hash, level, cur->size);
 3    void *entry = cur->array[pos];
 4    void *node = UNTAG(entry);
 5    if (node == NULL){
 6       if (insert_snode(key, val, hash, cur, pos))
 7          return SUCCESS;
 8       return insert_key(key, val, hash, level, cur, prev);        // try again
 9    } else if (IS_ANode(entry))
10       return insert_key(key, val, hash, level+1, (ANode)node, cur);
11    else if (IS_SNode(entry))
12       return collision_snode(key, val, hash, level, cur, prev, entry, pos);
13    else if (IS_ENode(entry))
14       complete_expansion((ENode)node);
15    else if (IS_CNode(entry))
16       complete_compaction(entry);
17    return FAILURE;
18 }
```

Listing 3.3: Pseudo-code for the insert_key() procedure

```
int insert_snode(size_t key, size_t val, size_t hash, ANode cur, int pos){
   SNode new_snode = alloc_init_snode(key, val, hash, NULL);
   new_snode = PUT_TAG(new_snode, SNODE);
   if (CAS(cur->array[pos], NULL, new_snode))
      return SUCCESS;
   free(UNTAG(new_snode));
   return FAILURE;
}
```

Listing 3.4: Pseudo-code for the insert_snode() procedure

One of the things we should consider is if the insert_key() algorithm should complete a pending compaction procedure since that will delay the insertion process. Compaction might increase the number of collisions, and we know that a node can be sooner inserted on that path. The only case that makes this decision be more efficient is when the node we are trying to insert has the same key like the one in the compaction

```
int collision_snode(size_t key, size_t val, size_t hash, int level, ANode cur,
    ANode prev, void *entry, int pos) {
   SNode snode = (SNode)UNTAG(entry);
   void *pending_entry = snode->pending;
   if (UNTAG(pending_entry) == NULL){                          // no pending changes
      if (snode->key == key) {                         // same key, try to update value
         SNode new_snode = alloc_init_snode(key, val, hash, NULL);
         new_snode = PUT_TAG(new_snode, SNODE);
         if (CAS(snode->pending, NULL, new_snode)) {    // annouce pending update
            CAS(cur->array[pos], entry, new_snode);               // perform update
            return SUCCESS;
         }
         free(UNTAG(new_snode));
         return insert_key(key, val, hash, level, cur, prev);
      } else if (IS_Narrow(cur))       // different keys and narrow ANode, expand
         expansion(hash, level, cur, prev);
      else {                      // different keys and wide ANode, insert new level
         ANode new_anode = alloc_init_anode_snode(snode->key, snode->val,
             level+1);
         if (CAS(snode->pending, NULL, new_anode))  // annouce pending new level
            // perform insertion of new level
            CAS(cur->array[pos], entry, new_anode);
         else {
            free(UNTAG(new_anode));
            return insert_key(key, val, hash, level, cur, prev);
         }
      }
   } else if (IS_ANode(pending_entry) || IS_SNode(pending_entry)) {
      CAS(cur->array[pos], entry, snode->pending);
      return insert_key(key, val, hash, level, cur, prev);
   }
   return FAILURE;
}
```

Listing 3.5: Pseudo-code for the collision_snode() procedure

procedure. Since the insert algorithm provides lock-freedom, any operation will complete after a finite number of steps. In this case, the number of steps depends on the paths that threads will follow. It is possible that a thread does not find a pending operation and, if that happens, the thread should complete normally its own operation.

3.1.3 Remove

The remove algorithm follows the same fundamentals of the insert algorithm. Listing 3.6 shows the pseudo-code for the remove_key() procedure that implements the remove algorithm. It receives the key to be removed and the respective hash, the level which it is traversing, the current and the previous ANode

being traversed. Like with the insert, the remove restarts in case of failure until it reaches a successful result.
If reaching an empty entry, like in lookup, it means that there is no node in the data structure with the key we are searching for. In such a case, it returns with success (lines 5-6). Since the remove algorithm helps all pending operations, we have the guarantee that there are no nodes with that key in pending updates.
If the algorithm reaches an array entry referring to a SNode, a different procedure will be followed (lines 12-13) as shown in Listing 3.7 that represents the remove_snode() procedure. Regarding the remove_snode() procedure, consider that SNode holds a different key from the one we are trying to remove, that key is not present on the structure and, as a result, the algorithm succeeds, because that key does not exist (lines 3-4). The same occurs if we reach an empty array entry in remove_key() . Otherwise, if the considered SNode pending field is not empty, indicating pending updates, that thread should help the operation to complete like explained for the insert algorithm (lines 13-15) and, after that, restart and try again to remove the key. The same procedure is taken in the case of found an ENode, indicating an expansion procedure, or a CNode completing a compaction procedure (lines 14-18). In case the considered SNode is the one that we are trying to remove (lines 6), and no pending operations are affecting the SNode, so we are able to remove it. For that is used a CAS directive in the previous array entry (line 7). Next, we need to check if the pending field remains empty since it is possible the node to change into a frozen state between checking the pending field and the CAS directive. If so, the algorithm reenters in order to try to catch replications of the node that was de-referenced (lines 8-9). If no replications are found, the algorithm ends up on an empty entry or on a SNode with a different key. If found a replication it proceeds in order to remove that one. Every time we remove a node, we check if the previous array is in a compaction state, i.e, having just one non-empty entry, (this happens on remove_key() procedure, lines 8-11). If so, we can delete that ANode and try to compact the data structure. The compaction state is checked through all arrays on the path we performed the removal. This guarantees that the structure maintains as few levels as possible, reducing the traversal complexity and cost.

The only case where the remove procedure completes with "out-dated" information is when comes from a frozen empty entry. We do not check the updated state of the structure or even help to complete pending changes, because the only way for that entry become occupied is if another thread completes an insertion of that node. But even there, the insertion only occurs after the changes are concluded, so it is safe to assume that the data structure does not has that key assigned.

3.1.4 Expansion

The goal of the expansion procedure is to replace a narrow ANode (4 slots) with a wide one (16 slots), i.e., change the small array into an array that represents the full level capacity for that path. This procedure is called as a result of a collision situation and the current level is represented by a narrow ANode. Listing 3.8 shows the pseudo-code for the expansion() procedure that implements the expansion algorithm. It receives the hash of the operation calling it, the level where it happened, and the corresponding current and previous ANode. As a result, it returns the wide ANode that has replaced the narrow one.
The algorithm starts by calculating the previous position, which represents the entry where the narrow ANode is referred (line 2). Then, it creates an ENode in the thread's local memory (line 3). This ENode serves to signal the other threads that an expansion is happening and provides the needed information to the other threads be able to help to complete the procedure. The ENode also permits that threads access

```
int remove_key(size_t key, size_t hash, int level, ANode cur, ANode prev){
   int pos = GET_POS(hash, level, cur->size);
   void *entry = cur->array[pos];
   void *node = UNTAG(entry);
   if (node == NULL)
      return SUCCESS;
   else if (IS_ANode(entry)) {
      int status = remove_key(key, hash, level+1, (ANode)entry, cur);
      if (status && prev != NULL && is_compactable(cur))
         compaction(hash, level, cur, prev);
      return status;
   } else if (IS_SNode(entry))
      return remove_snode(key, hash, level, cur, prev, (ANode)entry, pos);
   else if (IS_ENode(entry))
      complete_expansion(node);
   else if (IS_CNode(entry))
      complete_compaction(node);
   return FAILURE;
}
```

Listing 3.6: Pseudo-code for the remove_key() procedure

```
int remove_snode(size_t key, size_t hash, int level, ANode cur, ANode prev,
    void *entry, int pos){
   SNode snode = (SNode)UNTAG(entry);
   if (snode->key != key)                          // diferent key, nothing to remove
      return SUCCESS;
   void *pending_entry = snode->pending;
   if (UNTAG(pending_entry) == NULL){              // same key and no pending changes
      if (CAS(cur->array[pos], entry, NULL)) {
         if (IS_FNode(pending_entry))                 // node froozen in the meantime
            return FAILURE;
         return SUCCESS;
      }
      return remove_key(key, hash, level, cur, prev);
   } else if (!IS_FNode(pending_entry))                          // pending change
      // perform pending change
      CAS(cur->array[pos], entry, snode->pending);
   return FAILURE;
}
```

Listing 3.7: Pseudo-code for the remove_snode() procedure

the old narrow ANode in order to perform lookups and complete ongoing changes. In the continuation, we are able to update the entry of the previous array with a CAS to point to ENode, which contains the reference to the new wide ANode (line 5). If CAS fails, that means that some change happened on this node and probably the expansion is already ongoing or will not be needed. In such cases, the procedure tries again (line 10). After ENode is settled, all other threads that travel through that path will be able to complete the expansion procedure. To do that, threads call the complete_expansion() sub-routine, that is shown in Listing 3.9. The complete_expansion() procedure consists of freezing all the array entries in the narrow ANode held by ENode, in order to stop further changes (line 2). When all entries are frozen, they are copied and re-mapped in a new wide ANode (line 3). Entries will not collide since the mapping is done by bits, and basically each previous position is now mapped into four (4*4=16). At this moment, it assigns the new ANode into the ENode, providing to all running threads the knowledge that this step of expansion is completed. If CAS fails, meaning that another thread has completed the procedure, it reads the wide ANode that is committed on ENode (lines 4-6). Finally, the new ANode is set in the previous entry that refers to ENode (line 10).
Figure 3.4 illustrates how this procedure is done for a data structure with two ANodes levels and two SNodes.

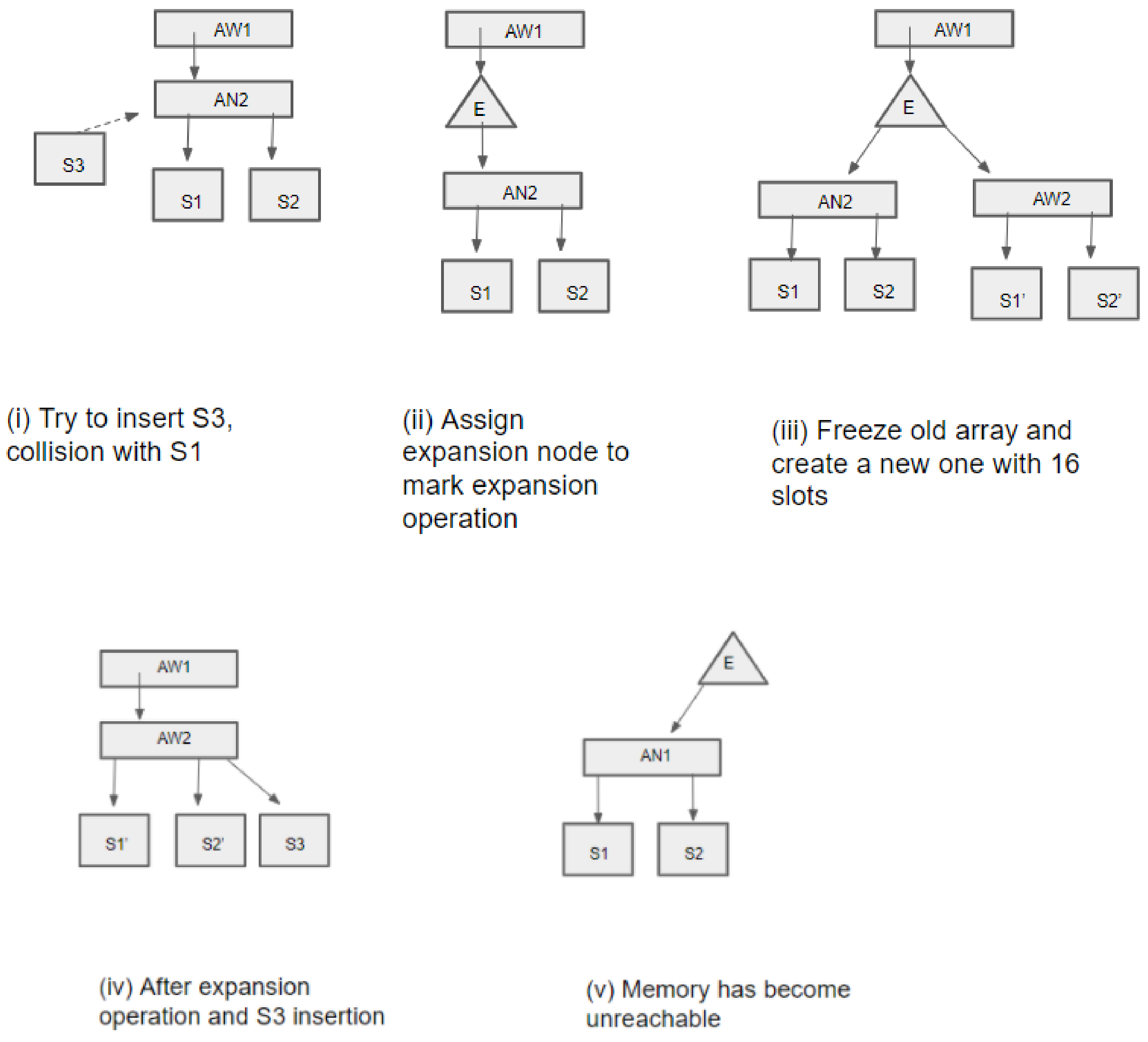
AW1
AN2
S3
S1
S2
(i) Try to insert S3, collision with S1
AW1
E
AN2
S1
S2
(ii) Assign expansion node to mark expansion operation
AW1
E
AN2
AW2
S1
S2
S1'
S2'
(iii) Freeze old array and create a new one with 16 slots
AW1
AW2
S1'
S2'
S3
(iv) After expansion operation and S3 insertion
E
AN1
S1
S2
(v) Memory has become unreachable

Figure 3.4: Expansion in CTries

As we can see in Fig. 3.4, this process turns the previous ANode unreachable, which will require some form of memory reclamation. This will be discussed further in section 4.

3.1.5 Compaction

Compaction is used to remove levels from the data structure. This characteristic of being resizable allows us to save memory space and reduce the number of traversal steps to reach the stored nodes. This procedure may be unseemly from a performance point of view since like the expansion it introduces overheads. However, it can reduce significantly the memory space usage. Listing 3.10 shows the pseudo-code for the compaction() procedure that implements the compaction algorithm. The compression operation is very similar to the expansion one, and like there, it includes a complete_compaction() that allows other threads to help to complete the compaction procedure. Like the expansion, the compaction receives the

```
ANode expansion(size_t hash, int level, ANode cur, ANode prev){
   int prev_pos = GET_POS(hash, level-1, prev->size);
   ENode enode = alloc_init_enode(cur, NULL, prev, prev_pos, level);
   void *enode_entry = PUT_TAG(enode, ENODE);
   if (CAS(prev->array[prev_pos], cur, enode_entry)){  // assign enode to ctrie
      complete_expansion(enode);
      return enode->wide;
   }
   free(enode);                                          // CAS failed , free enode
   return cur;
}
```

Listing 3.8: Pseudo-code for the expansion() procedure

```
void complete_expansion(ENode enode){
   freeze(enode->narrow);
   ANode wide = alloc_init_wide_anode(enode->narrow, enode->level);
   if (!CAS(enode->wide, NULL, wide)) {              // if wide already assigned
      free(wide);
      wide = enode->wide;                                          //read wide
   }
   int prev_pos = enode->prev_pos;
   void* enode_entry = PUT_TAG(enode, ENODE);
   CAS(enode->prev->array[prev_pos], enode_entry, wide);
}
```

Listing 3.9: Pseudo-code for the complete_expansion() procedure

hash, the level, the ANode corresponding to that level, and the previous ANode that was traversed. This procedure ends up in a successful or failure state.
The compaction routine starts by calculating the previous position holding the current ANode (line 2). After creating the CNode (that will mark the operation), it follows the same principle as in expansion procedure, to mark that procedure is happening, by getting the newly created CNode with a CAS in the previous entry referring ANode (line 5). Now, all threads can call the complete_compaction() subroutine, to complete the compaction procedure. The complete_compaction() is represented in Listing 3.11. It starts by freezing all the array entries, which should be mainly composed by empty entries (line 3). This allows other threads to know that an operation is running on that ANode and prevent further changes. When all the array entries are frozen and are guaranteed that no updates will happen on that array, we go through all entries (lines 5-6) and if a SNode is found, we save that reference to further replicate the node (lines 7-8). If another SNode is found that means that insertion occurred between the start of the compaction procedure and of freezing, and ANode is not anymore in a compaction state. In such cases, a copy of ANode has been created on thread local memory (lines 16-17). Threads finish the procedure by setting the local SNode or the copied local ANode with a CAS in the previous entry referring CNode (line 20). In both cases, threads have all the needed information to complete the operation. To better understand this procedure, Fig. 3.5 shows an example for the data structure state during a compaction procedure.

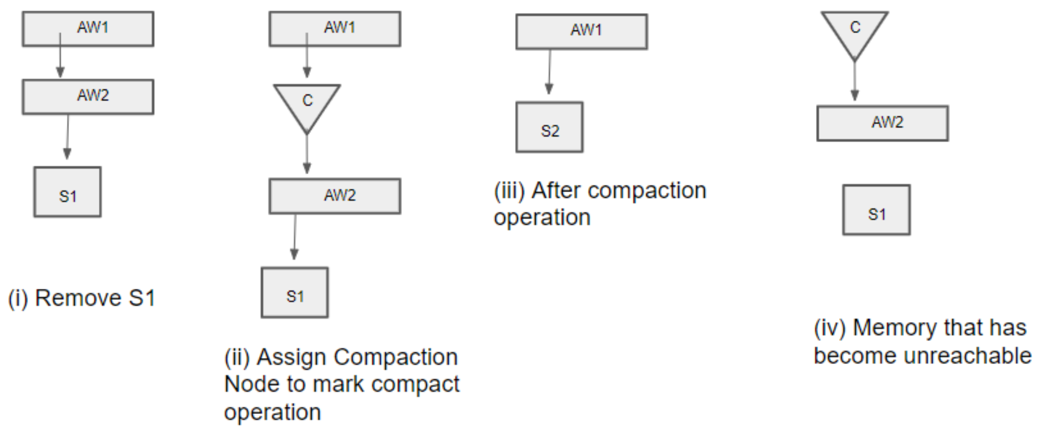
AW1
AW2
S1
(i) Remove S1
AW1
C
AW2
S1
(ii) Assign Compaction Node to mark compact operation
AW1
S2
(iii) After compaction operation
C
AW2
S1
(iv) Memory that has become unreachable

Figure 3.5: Remove and Compaction in CTries

```
int compaction(size_t hash, int level, ANode cur, ANode prev){
   int prev_pos = GET_POS(hash, level-1, prev->size);
   CNode cnode = alloc_init_cnode(cur, prev, prev_pos, level);
   void *cnode_entry = PUT_TAG(cnode, CNODE);
   if (CAS(prev->array[prev_pos], cur, cnode_entry))
      return complete_compaction(cnode);
   free(cnode);
   return FAILURE;
}
```

Listing 3.10: Pseudo-code for the compaction() procedure

3.1.6 Freeze

Freezing a SNode is done by marking it's his pending field as frozen. On the other hand, an ANode is frozen by creating a frozen node (FNode) that holds a reference to the frozen ANode. Freezing a node prevents subsequent updates by other threads on the entry holding that node. Like on the other procedures, steps are different depending on the type of node. Listing 3.12 shows the pseudo-code for the freeze() procedure that implements the freeze algorithm for a given ANode. It should freeze the entire ANode to avoid further changes on that level. The freeze routine goes through all array entries of ANode and freezes the respective entries. On empty entries, it just tags the entry with an FNode tag (lines 6-8). This prevents other threads succeed doing the CAS directive. If found a SNode, first it checks if pending changes are set in the pending field. If so, it helps pending changes before freezing (lines 16-17). Although freeze presents a mechanism to stop future updates on the data structure, it needs to complete ongoing changes to guarantee that no changes will be lost meanwhile. Otherwise, if a pending field is empty, it

```
int complete_compaction(CNode cnode){
   ANode anode = (ANode)cnode->cur;
   freeze(anode);
   void *compact = NULL;
   for (int i=0; i < anode->size; i++) {
      void *node = anode->array[i];
      if (IS_SNODE(node) && (compact == NULL))
         compact = node;
      else if (UNTAG(node) != NULL){
         compact = anode;
         break;
      }
   }
   if (IS_SNode(compact)) {                           // only one SNode found
      compact = alloc_duplicate_snode(compact);
   } else if (IS_ANode(compact) { // at least two SNodes or another node found
      compact = alloc_duplicate_anode(compact);
   }
   void *cnode_entry = PUT_TAG(cnode, CNODE);
   CAS(cnode->prev->array[cnode->prev_pos], cnode_entry, compact);
   return ((compact == NULL) || IS_SNode(compact));
}
```

Listing 3.11: Pseudo-code for the complete_compaction() procedure

executes a CAS in order to mark that field as frozen (lines 13-15). After this CAS, we guarantee that other threads will be aware that SNode is frozen. On ANodes, we create an indirection to mark the frozen state, basically, we create an FNode that holds the ANode reference and set him in the previous ANode entry (lines 18-21). The algorithm reenters on that reference, which now marks an FNode, and recursively freezes all entries (lines 22-23). If a CNode entry is found, threads perform complete_compaction in order to complete any pending changes (lines 26-27).

Since freezing replaces ANode's entries one by one, it is possible that a thread adds a node during this process, if the corresponding entry is not yet frozen. However, this is not a problem, since the goal is to guarantee that no future updates will happen and not fresh updates. This behavior comes with a handoff when a thread performs the compaction procedure because it is possible that compaction will be not needed anymore. In such cases, the freeze routine completes all pending changes and retries to freeze that entry after. Since the algorithm helps all other pending changes before freezing, it guarantees that the frozen state will be mostly updated at that moment.

3.2 Problems

CTries achieve good performance and can be easily adapted to better accomplish program requirements. However, its implementation outside of a garbage collection environment, will not reclaim memory. The usage of memory can be exponential, leading the program to become unbearable for the system. Due to

```
void freeze(ANode cur){
   int i=0;
   while (i < cur->size) {
      void *entry = cur->array[i];
      void *node = UNTAG(entry);
      if (entry == NULL)                            // if NULL, tag it as frozen
         if (CAS(cur->array[i], entry, PUT_TAG(NULL, FNODE)))
            i++;
      else if (IS_SNode(entry)){
         SNode snode =(SNode)node;
         void *pending_entry = snode->pending;
         void *pending = UNTAG(pending_entry);
         if (pending_entry == NULL)        // SNode without pending changes, tag
             pending field
            if (CAS(snode->pending, NULL, PUT_TAG(NULL,FNODE)))
               i++;
         else if (!IS_FNode(pending_entry))                      // pending change
            CAS(cur->array[i], entry, snode->pending);// perform pending change
      } else if (IS_ANode(entry)){
          FNode fnode = alloc_init_fnode(NULL, entry, cur, i, cur->level+1);
          fnode = PUT_TAG(fnode, FNODE)
          CAS(cur->array[i], entry, fnode);
      } else if (IS_FNode(entry) && (node != NULL))
         freeze((FNode)node->frozen);
      } else if (IS_ENode(entry)){
         complete_expansion(node);
      else if (IS_CNode(entry))
         complete_compaction(node);
   }
}
```

Listing 3.12: Pseudo-code for the freeze() procedure

the dynamic size property and the immutable nodes of CTries, it can produce unreclaimable memory not only on remove operations. Indeed the insertion of just one node, in a situation that triggers an expansion procedure, makes the level obsolete, and all that nodes become garbage. Note that, in this section, when we are talking about removing a node it is not considered the reclamation operation of that piece of memory. That is because, in this design, we are assuming that memory reclamation is a task of the garbage collector. As discussed previously, this delegation can affect the lock-freedom property of the data structure as a whole.

This version of CTries also suffers from the ABA problem, and a memory reclamation scheme should fix this problem by design. If a reference is not reclaimed until it is guaranteed that no other thread is accessing it, the problem is solved, since the block will not be reused. Because of the persistent pointers problem, algorithms become limited in memory reclamation solutions. The restricted reuse (no reclamation) can not be used, because of the possibility of immediate reuse. Hazard pointers and other pointer solutions can not be used too, because they allow the reclamation of blocks that are indirectly reachable from private references. To use pointer based techniques, it is essential to solve the persistent pointer problem. The main problem with memory reclamation is to know if a node can be safely reclaimed, this is, have the knowledge that if at a given time it is possible to reclaim that piece of memory, i.e. if no other threads are accessing that memory position.

Chapter 4

CTries and Memory Reclamation

In this chapter, we describe our approach of applying the current state-of-art memory reclamation methods to the CTries data structure. We start by discussing the application of the HHL method and why it does not fit with our goals. Then, we present how we adapted it to the CTries. This new method is closely integrated with the base implementation and exploits the CTries structure to achieve optimal memory bounds.

4.1 State-of-the-Art Methods

Current reclamation methods usually rely on the remove operation in order to guarantee that removed nodes become unreachable. However, since the CTries removes nodes in several procedures than remove, that guarantee needs to be applied to various operations. As we mentioned early, hazard pointers can not be applied to the CTries without modifications due to the persistent pointers problem. On the other hand, methods that rely on nodes age, like hazard eras or interval based reclamation, can be applied since the reclamation procedure protects all the nodes inside that period of time. A satisfying memory reclamation scheme, should solve by design the ABA problem and provide bounded memory usage. To apply memory reclamation methods, the data structure needs to be adapted in order to:

- know when a node becomes unreachable;
- adjust all needed operations to provide that guarantee;

Because of the immutable property of CTries, we decided to adjust all needed operations, since once nodes are detached from the structure, they will not be assigned again, before they pass throw the reclamation process.
To adjust the CTries to memory reclamation methods, we need to change some aspects:

- To reclaim nodes:
  - Nodes need to be modified to include hash and level fields;
  - Adjust the traversing procedure to protect nodes;
  - Track the thread id during procedures;
- To solve the persist pointer problem:

  - When removing nodes, nullify pointers;
  - Algorithms need to be modified to deal with NULL pointers;

- To solve the ABA problem:
  - Adjust all operations to guarantee that the reclamation cycle is followed;

Every time a node is retired from the structure, due to an insertion or a remove, that node now passes through the reclamation procedure. This guarantees that the reclamation cycle is followed and that the ABA problem is solved. The first step of the reclamation procedure is to nullify all pointers of a node. This step is needed to solve the persistent pointer problem. When one node is removed from the structure, all child nodes are removed too, meaning that we need to add all child nodes to the reclamation list too. All methods should do these steps in order to be correctly adapted. The way that reclamation is controlled depends on the method itself. Methods will need to adapt the traversal procedure of algorithms in order to guarantee that nodes become protected before being accessed in the traversal step. Figure 4.1 shows the nodes life cycle for the new CTries with memory reclamation.

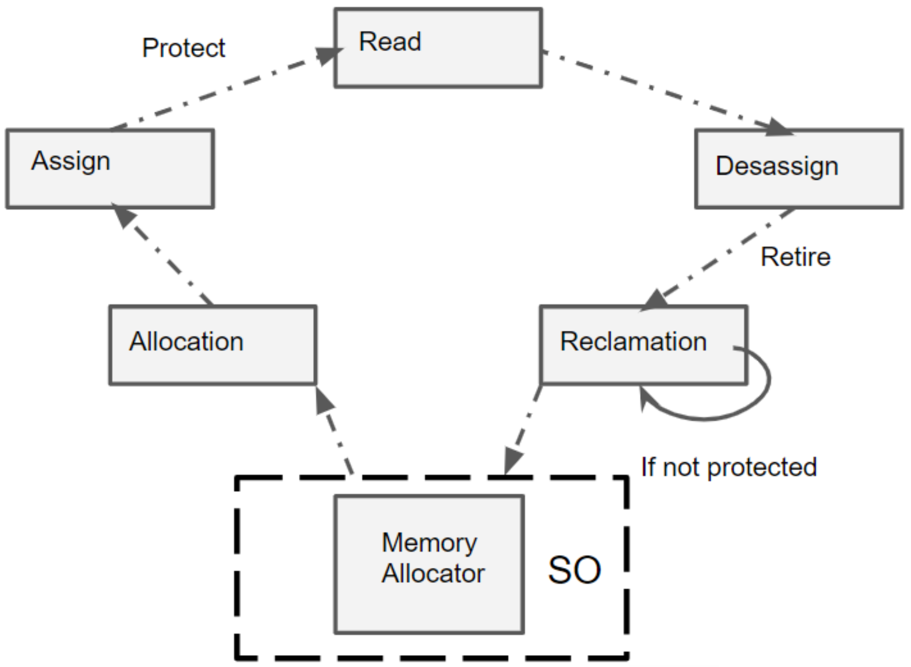

Figure 4.1: Nodes life cycle.

Nodes are initially created with normal allocation in threads local memory, and then they are assigned into the CTrie data structure. After that, nodes can be read by the other threads, but before reading a node, threads need to somehow protect the node. During their lifetimes, nodes can be later deassigned from the data structure for different reasons. After that, the node is retired, by nullifying all pointers to the other nodes. If a node is not protected, it can be safely reclaimed and its memory is returned to the OS. Different reclamation methods mainly vary in the way that they protect nodes.

4.2 Adaption of HHL

The LFHT and CTries data structures are both tree based and have two major groups of nodes, leaf nodes, and nodes with references to other nodes. In both LFHT and CTries, the nodes' locations are tracked by the hash and level fields. Each group of four bits in the hash represent the node position at each level. This similarity made the HHL reclamation method a good base option for memory reclamation in Ctries. However, HHL can not be directly adapted because the protection is only guaranteed to leaf nodes. In LFHT, there is no problem with that, since the structure does not vary so much and the hierarchy nodes are never removed. On the other hand, CTries is constantly allocating new nodes and replacing a significant amount of intermediate nodes, becoming essentially important the reclamation of this type of nodes.
As we saw, to implement memory reclamation we need to change the traversal procedure of data structure, to ensure that a node is protected by a hazard pair before accessing it. In LFHT, the protection only occurs in leaf nodes, since the level where a thread is in the data structure is only updated when reaching a leaf node. However, in CTries, all nodes need to be protected to consequently be reclaimed. So, in the CTries context, a single hazard pair may protect more than one node. For example, ANodes need to stay protected while any child SNode is protected. To guarantee that protection, algorithms need to update the thread's level every time they move in the data structure, instead of just when finding leaf nodes. Since otherwise, the following problem shown in Fig 4.2 can happen. Consider that thread T1 is inserting a leaf node S2 and stops on AW2 before updating its level. At this moment, another thread T2 removes the last SNode in AW2 and AW2 enters in a compaction situation. T2 successfully performs compaction and AW2 goes through the reclamation procedure. The hazard pair hash/level of T1 is (S2, 0) , which does not protect AW2. So, if AW2 is reclaimed, when T1 wakes up it is referring to invalid memory and may crash.

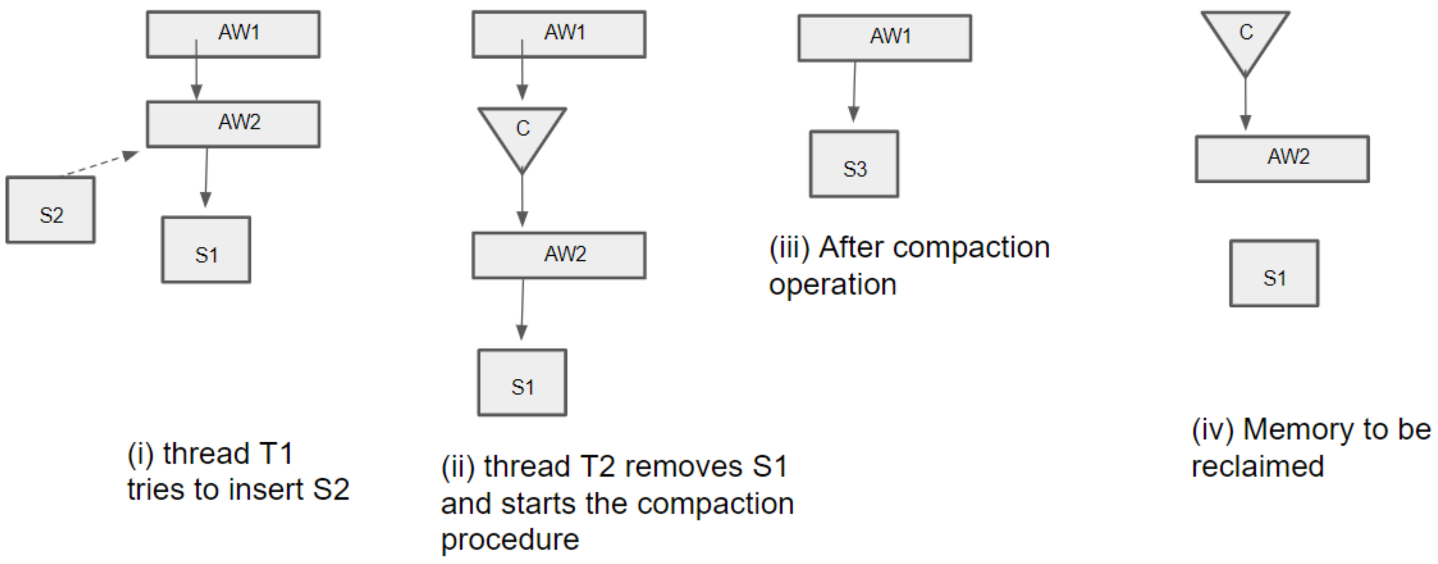
AW1
AW2
S2
S1
(i) thread T1 tries to insert S2
AW1
C
AW2
S1
(ii) thread T2 removes S1 and starts the compaction procedure
AW1
S3
(iii) After compaction operation
C
AW2
S1
(iv) Memory to be reclaimed

Figure 4.2: Problem if not updating the level as a thread moves in the data structure

As mentioned before, the reclamation procedure of CTries needs to be able to reclaim different types of nodes. In order to do that with HHL, it is needed to adapt to the reclamation rule. Nodes can only be reclaimed if the hash of a node does not match (until the node's level - 1) with the hazard pairs of any thread. We need to protect all nodes of the same level due to the procedures of CTries that traverse all layers of a level.
Another problem is that a direct application of HHL does not guarantee a bounded memory usage, as Fig. 4.3 illustrates. If thread T1 is traversing AN2, the hazard pair protects AN2 from reclamation. The problem appears if another thread T2 performs an expansion from AN2 to AW2. T2 will not reclaim AN2 because T1 is protecting it. That is correct, however, if T2 continues performing operations and now AW2 is in a compaction situation when T2 removes AW2 and tries to reclaim it, the hazard pair of T1 (that is still sleeping in AN2), matches with the hazard pair that protects AW2, and thus AW2 will not be reclaimed, this is the care although there is no thread referring AW2 and thus it can be reclaimed. The problem is that due to the resizable property of CTries, a hazard pair (hash, level) is not unique to each node, and can block all future nodes in the same level.

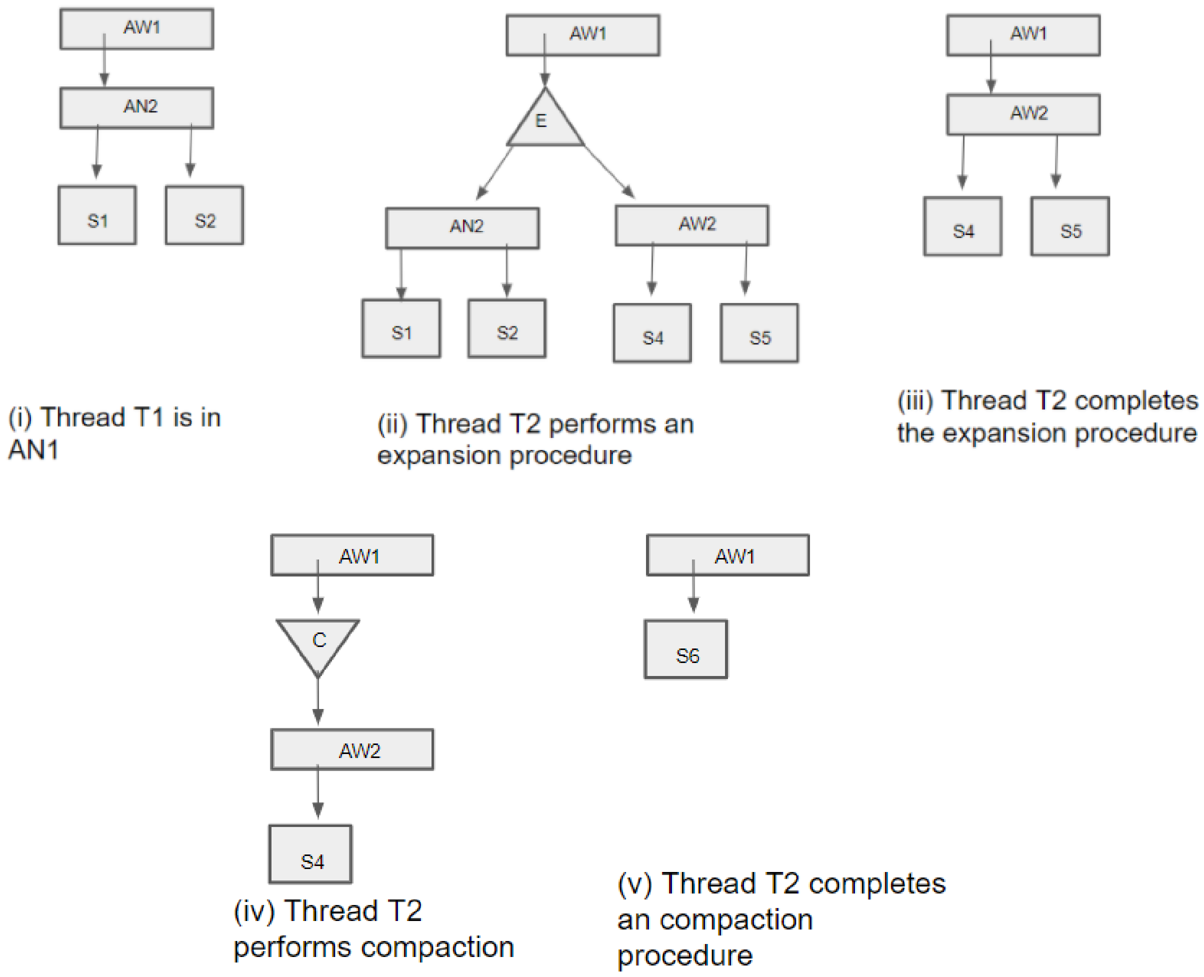
AW1
AN2
S1
S2
(i) Thread T1 is in AN1
AW1
E
AN2
AW2
S1
S2
S4
S5
(ii) Thread T2 performs an expansion procedure
AW1
AW2
S4
S5
(iii) Thread T2 completes the expansion procedure
AW1
C
AW2
S4
(iv) Thread T2 performs compaction
AW1
S6
(v) Thread T2 completes an compaction procedure

Figure 4.3: Memory unbounded problem

To guarantee bounded memory usage, we need to be able to track the difference between two equals hazard pairs at the same level.
To achieve a bounded memory reclamation solution, and implement a satisfying solution, we have modified the HHL method in the following way:

- all Nodes have been modified to include a blocking list;
- a new global array "Stuck Array", marks threads on outdated states;
- a new per thread clock num_op is used to distinguish equals hazard pairs over time;

Listing 4.1 shows the changes done to the data structures that compose CTries, and Figure 4.4 shows how the new structures , defined on Listing 4.2, are organized to implement our solution.

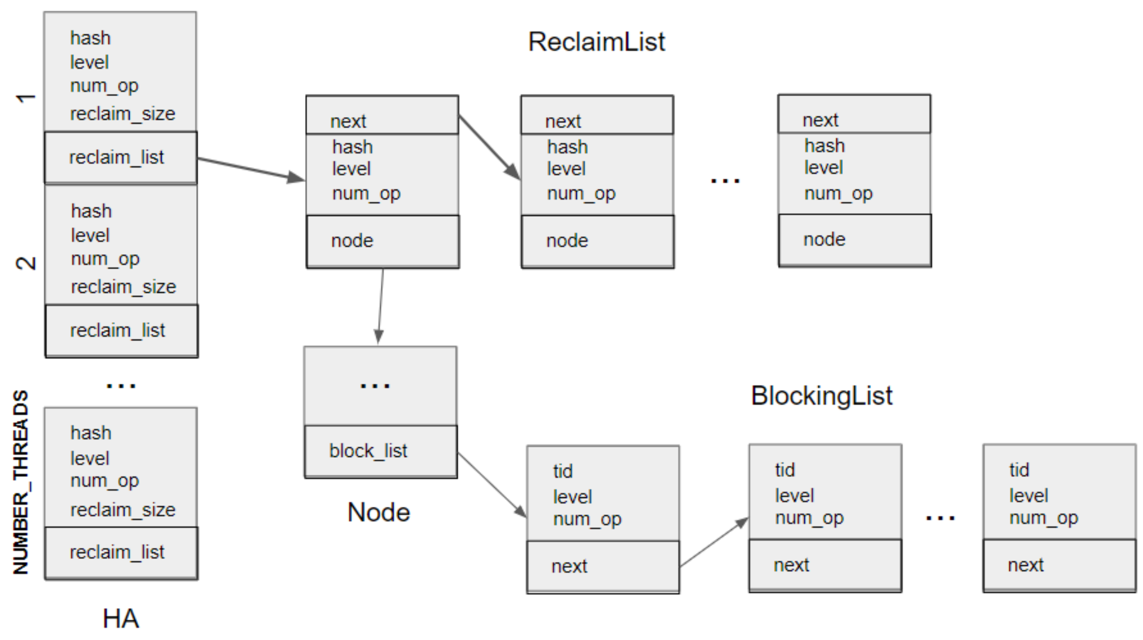

Figure 4.4: Structs that implement Reclamation Solution

4.2.1 Main Idea

As we mentioned before, the application of HHL may result in unbounded memory usage. To solve that, it is crucial to distinguish between two equals hazard pairs that protect different nodes. This can be achieved in different ways, for example, it is possible to distinguish equals hazard pairs of different nodes following a pointer based solution, like adding a pointer to the current node in the hazard pair. However, pointer based solutions add some unwanted overhead since in every traversal step the pointer needs to be updated. Our approach uses a per thread clock "num_op" that tracks the number of times that a thread has entered the data structure by the root reference, i.e, this clock is incremented every time a thread passes through the root reference. Since the clock is local to each thread, it allows us to track if a certain thread remains stopped or is traversing new nodes. Note however that this clock is not a global

```

typedef struct {
    size_t key;
    size_t val;
    void *pending;
    size_t hash;
    int level;
    BlockingList block_list;  //hold threads that block node from reclamation
} *SNode;

typedef struct {
    void **array;
    int size;
    size_t hash;
    int level;
    BlockingList block_list;  //hold threads that block node from reclamation
} *ANode

typedef struct {
    ANode narrow;
    ANode wide;
    ANode prev;
    int prev_pos;
    int level;
    size_t hash;
    BlockingList block_list;  //hold threads that block node from reclamation
} *ENode

typedef struct {
    ANode cur;
    ANode prev;
    size_t hash;
    BlockingList block_list;  //hold threads that block node from reclamation
} *CNode

typedef struct {
    ANode cur;
    ANode frozen;
    ANode prev;
    int prev_pos;
    int level;
    size_t hash;
    BlockingList block_list;  //hold threads that block node from reclamation
} *FNode
```

Listing 4.1: Changed structures.

```c
typedef struct {
   int tid;
   size_t num_op;
   int level;
   BlockingList next;
} *BlockingList

typedef struct {
   void *node;
   size_t hash;
   int level;
   size_t num_op;
   ReclaimList next;
} *ReclaimList

typedef struct hazard_entry {
   size_t hash;
   int level;
   size_t num_op;
   int reclaim_size;
   ReclaimList reclaim_list;
} HazardEntry;

typedef HazardEntry* HazardArray;

HazardArray HA;                                           // shared hazard array
HazardArray LOCAL_HA, LOCAL_HA2;         // two local private array per thread
HazardEntry** STUCK_HA;                // to keep track of the suspended threads
```

Listing 4.2: New structures.

clock, so it is not possible to distinguish between two equals hazard pairs, although this provides sufficient information to make it possible, as we will explain later. In order to guarantee that a node will not be blocked from reclamation by threads who have the same hazard pair but have entered after the node was retired from the structure, nodes in a reclamation list now have an associated blocking_list. The goal of this list is to mark the threads that, after the removal of the node from the structure, are blocking that node from being reclaimed. This also guarantees that threads with equal hazard pairs will not interfere with the reclamation of a node that is out of structure at the moment they have entered the structure. A Blocking list is set when nodes pass through the retirement process. As said, the goal of blocking_list is to mark the threads that can hold references to the newly retired nodes. Once the Blocking list is only set after nodes be retired from the structure, threads do not have complications in time of set blocking_list, because only that thread do that work. The blocking list contains the tid (thread id), level, and num_op of the threads that can have a reference to the node. This guarantee that threads that will collide in the same hazard pairs in the future will not interfere in the reclamation process of a node already removed. The threads that can block a node from reclamation are all marked in the node's blocking_list. So, at the time of reclamation, we only need to consider those threads. If the current hazard pair of a thread is the same as the one saved on the node's blocking_list, the thread is still blocking that node from reclamation. If not, that means that the thread has moved and is no longer blocking the node from reclamation. For example, in the expansion routine, when an expansion node is removed, the expansion node and the old narrow ANode, pass through the retirement procedure that creates the node's blocking_list, this way threads that enter on the new wide node after the retirement procedure will not block the reclamation of that nodes.

However, in the end, this solution is still unbounded, since threads that stop before a node is created are still blocking the reclamation of that node. Following Fig. 4.3, consider that thread T1 stops in AN2, meaning that its hazard triplet is (hash=h1, level=1, num_op=1). Meanwhile, thread T2 performs expansion from AN2 to AW2. When T2 retires AN2, T1 is added to its blocking_list. At this point, there is no problem since T1 actually blocks the reclamation of AN2. However, when AW2 goes through retirement, T1 will block its reclamation too. Indeed, T1 will block the reclamation of all future nodes in that level, because threads do not know if T1 is traversing the node or remains stopped in old nodes. In order to know if a thread is stuck before a node is assigned to the structure, we use a global array named (stuck array). The stuck array is a global array, multi-writer multi-reader, that holds immutable hazard triplets (hash, level, num_op). The goal of the stuck array is to mark the threads that stay stopped in old outdated nodes, that are no longer part of the CTrie data structure.

In order to get bounded memory, we update the stuck array when inserting a new level, in order to mark the threads that are still in the old ANodes that exist on that level. When an ANodes passes through the compaction procedure, threads that are in the stuck array and remain with the same hazard pair, and the hazard pair matches with the node hazard pair, that thread will not be added into the blocking_list of that ANode. This guarantees that when an ANode was inserted, the stopped threads are marked on the stuck array, if at the moment of ANode retirement, the threads num_op is the same as in the stuck array, then that thread stay stopped the entire life of ANode and should not interfere with the reclamation of that ANode.

The stuck array only cares about new levels, we do not need to mark the threads that are stuck between expansion. This is because, the ENode used in the expansion procedure, holds a reference to both the narrow and wide ANodes. These two hazard triplets will not be distinguished, since threads in a narrow ANode should also protect wide ANode. This indirect control of the compaction procedure is enough to achieve a bounded solution.

With this solution, when a thread is inserting a new level ANodes (narrow), it starts by reading the hazard triplets of all threads, keeping a copy of them. After that, it assigns the narrow ANodes to the data structure and checks if any thread is in the position of the newly added node. If a thread has the same hazard triplet as in copy before the node was assigned, it is impossible that the thread sees the newly added node. In such a case, the threads on that position will become outdated, since a new node was entered on that position. The stuck array is updated if copies are equal, i.e., the hazard pairs match with each other and the num_op is the same as the HA num_op. Therefore marking that these threads are in an old position. To update the stuck array we re-check if the hazard triplets are still the same, guaranteeing that threads do not move meanwhile, by updating the stuck array with CAS. The update to the stuck array only happens if num_op present there is lesser than the new one.

However, if some thread is stopped on one level, the thread will block all nodes at that level. If nodes are inserted and removed from that level without trigger compaction, the thread block an infinite amount of nodes. To guarantee that this never happens, we force a replacement of that level. During replacement, the Stuck array is updated and nodes in the new level will not be blocked by that thread. This forced replacement induces the threads that are in that level into an outdated state.

To better show how this solution achieves bounded memory, consider the example in Fig. 4.5. The Figure represents different states of the Ctries where new levels are created and removed from the structure. Consider that, in the state (i), threads T1 and T2 are traversing narrow ANode AN2, and no other thread is on that path. At this point, the hazard array of T1 and T2 protect AN2. Somewhere between state

(i) and (ii), AN2 passes through an expansion procedure, and as explained before, the reclamation will not distinguish between threads that protect AN2 or threads that protect the new AW2, this is because during expansion threads need to protect both nodes since both are reachable from the expansion node. Consider now that, T2 has performed this expansion and starts the reclamation procedure of ENode and AN2. When AN2 enters the reclamation, his blocking_list is set. At this point, the stuck array is empty, so T1 is added to the blocking_list since it is protecting the node. Then consider that, T1 removes a SNode and AW2 becomes compactable, and T1 starts the compaction procedure. At state (ii), T2 is stopped and its hazard array is still protecting AW2. In the compaction routine, when nodes pass through the retirement process, if a thread is protecting nodes in the hazard array, and the hazard triplet is not in the stuck array, the thread is marked in the node's blocking_list. If the stuck array has the same hazard triplet, we do not mark that thread on the node's blocking_list. This way, the blocking_list of AW2 is T1 hazard pair. In state (iii), T1 and T2 remain stopped, and T3 reads S2. Between state (iii) and (iv), T4 inserts a new level (AN3). Before inserting AN3, it reads the hazard array, assigns AN3, and reads again the hazard array. If both reads are the same and match to that node, that threads are in outdated states. To mark that, we update the stuck array with that hazard pairs. Remember that, the stuck array is only updated if num_op of hazard triplet is greater than the one on the stuck array. At that point of state (iv), the stuck array has the hazard triplet of T1 and T2. T3 does not enter in the stuck array because it is in the previous level. After, if AN3 enters on a compact state (iv), reclamation is done like in state (ii). When T4 retires CNode, blocking list of AN3 and other nodes on that level, only add threads that hazard array match and is not in the stuck array. In this case, T1 and T2 will not block nodes of that level from reclamation. In state (vi), when node S4 is retired, thread T3 will block its reclamation, since the hazard array of T3 protects S4 too. This is needed because T3 can be referring to S2 or S4. In state (vii), AW1 was replaced by the AW7, this way, T3 is updated on the Stuck array and will not block the reclamation of S5.

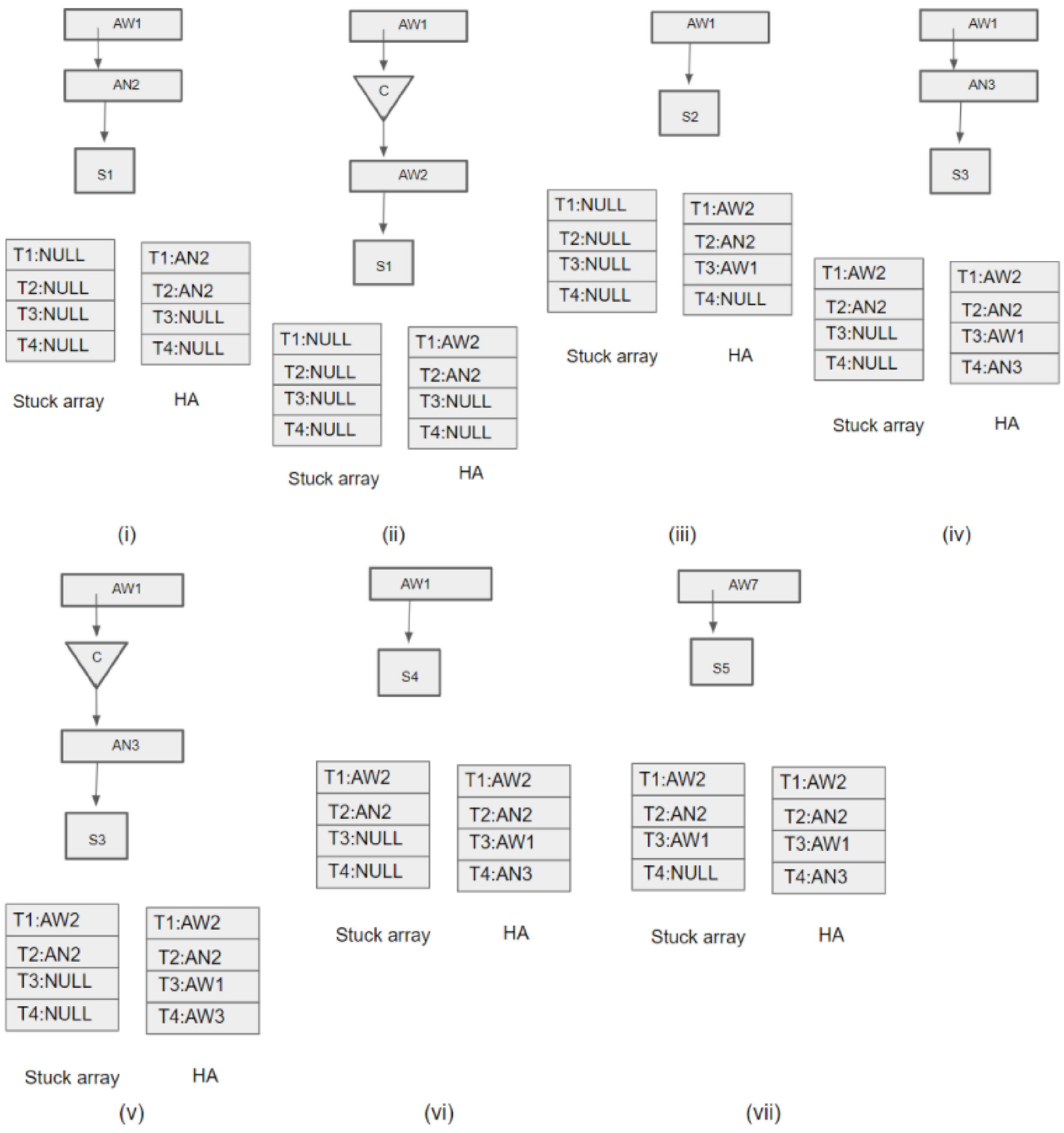

Figure 4.5: Bounded Reclamation Solution

This guarantee that stopped threads will not block new nodes from reclamation. This solution maintains the reclamation cycle show in Fig. 4.6 and guarantees that the ABA problem does not happen.

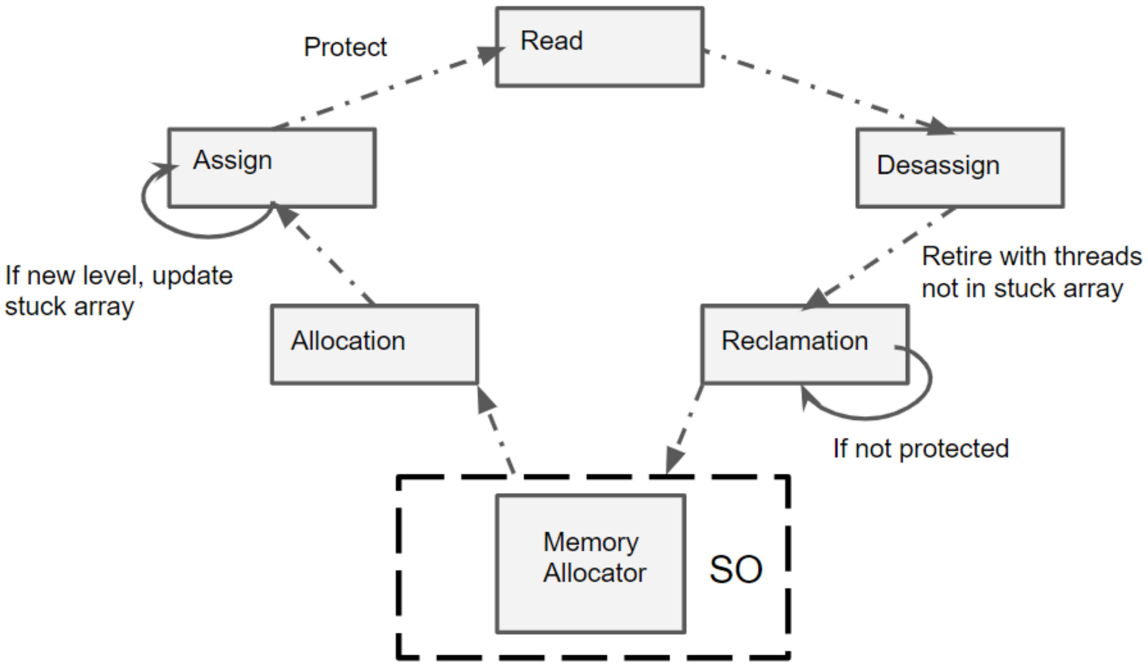

Figure 4.6: Bounded Reclamation Scheme Cycle

On other hand, the Blocking list guarantees that when a node will be reclaimed, threads that are traversing the structure after the node's lifetime will not interfere in the reclamation of that node. On other hand, the stuck array guarantees that when a node will be reclaimed, threads that were traversing the structure before the node's level lifetime, will not interfere in the reclamation of that node. In the end, only threads that are traversing the structure during the node's level lifetimes will be considered in the reclamation procedure.

4.3 Limitations

Our solution to the memory reclamation problem is an adaptation of the HHL method to the Ctries, which can work in data structures that present a tree based hierarchy. Generally, if the data structure can map nodes into a pair of hash and level, our method should work. Depending on the data structure properties, the method can even be optimized.
However, this method only works inside some restrictions. Since we use a num_op value to track the different operations in the same thread, the solution only works correctly if num_op does not overflow. In our implementation, since we are using size_t variable, this puts a limit per thread operations at the maximum of 0xffffffffffffffff (64 bits). If this maximum is reached, it will overflow and the stuck array will not update properly and the bounded memory is not guaranteed. Another limitation is the size of the hash value (again of size_t). Some implementations of Ctries lead with hash collisions by attaching nodes in a list on the last level, thus the direct application of this method in that situation will preserve the bounded memory usage since one thread stuck on the last level only blocks that list from reclamation.
This method is not depending on how much nodes are on one level, and on how much levels the data structure has, so there are no limitations on that. However, this directly influences the bound of memory

usage and performance.

4.4 Guarantees

Our solution provides a memory bounded usage for the Ctries data structure without the loss of the lock-free properties. The memory bound is achieved by hazard pairs (hash, level, num_op) and by tracking the threads inside the structure. To define the memory bound limit, we need the following variables:

- the number T of threads;
- the maximum number N of nodes in a single level;
- the threshold TE for invoking the reclamation procedure.

This way, our solution presents the following memory bounded equation:

$$T \times max(TE, N \times T) \tag{4.1}$$

4.5 Algorithms

4.5.1 Reclaim Node

The reclaim_node() procedure is responsible for the application of the memory reclamation method. Listing 4.3 shows the pseudo-code for the reclaim_node() procedure. This procedure runs right after a node is being removed from the data structure.
The first step of this procedure is to call the retire_node() procedure (line 2). Nodes are then added to a linked list (reclaim_list), this list is local to each thread (line 3). In other words, each thread reclaims the nodes that it has previously removed. Because the reclamation procedure is in some way expensive, nodes are only reclaimed after the reclamation list exceeds some threshold in the number of nodes (line 4). If the threshold is reached, we read all the hazard pairs from the global hazard array (line 5). Next, we go through all the reclaim_list, and on each node, we check if the node is in condition to be reclaimed with check_blocking_list (lines 6-18). If check_blocking_list returns SUCCESS, the node is reclaimed. Otherwise, if returns FAILURE, that node cannot be reclaimed. In that case, we check how many SNodes the reclamation list has at the same level, by the check_nodes() procedure. If the number of SNodes (in the same level and path) in the reclamation list passes a threshold, return as SUCCESS and the force_replacement() is called (lines 14,15). The force_replacement() routine follows the same idea as the expansion, and assign a copy of the current ANode.

The blocking_list is set in add_to_reclaim_list procedure, as shown in Listing 4.4, when nodes are already detached from the structure.

4.5.2 Retire Node

The retire_node procedure, as shown in Listing 4.5, is called every time one node is removed from the data structure. The goal of this procedure is to nullify the pointers present on a node, in such a way that

```
void reclaim_node(void *node_entry, int tid ){
   retire_node(node_entry, tid);
   add_to_reclaim_list(node_entry, tid);
   if (HA[tid].reclaim_size >= RECLAIM_THRESHOLD) {     // start reclamation
      copy_ha(LOCAL_HA, tid);                            // copy hazard pairs
      ReclaimList *ptr = &(LOCAL_HA[tid].reclaim_list);
      while (*ptr) {           // go through reclamation list and free nodes
         if (check_blocking_list((*ptr)->node, tid)) {
            ReclaimList *tmp = *ptr;
            *ptr = (*ptr)->next;
            free_reclaim_list_structure(tmp);
            HA[tid].reclaim_size--;
         } else {                                          //go to next node
            if (check_nodes(ptr))     //check if need to force a replacement
               force_replacement(ptr);
            ptr = &((*ptr)->next);
         }
      }
   }
}
```

Listing 4.3: Pseudo-code for the reclaim_node() procedure

threads that can be in that node will not continue traversing it and will be forced to reenter from the root node. The nullify of pointers informs those threads that they are in outdated versions of the structure. The nullify of pointers also block threads from removing the same nodes again, thus avoiding the double reclamation of nodes. Due to the immutable property of the data structure, when a node is removed, all the nodes reachable from that node (i.e., lower in the hierarchy) are removed too. This means that, when a node is added to the reclaim list, other nodes might be added too.

4.5.3 Check blocking list

The goal of this procedure, as shown in Listing 4.6, is to tell the reclaim procedure if a node can be reclaimed or not. This method starts by reading the head of the blocking list of a specific node (line 2). Then, it goes through the list, and for each thread associated with the current block, it checks if its hazard pair is still the same, which means that the thread does not advance meanwhile (line 8). And the node can not be reclaimed, returning as a failure (lines 9-10). If the blocking list of one node is empty, or all blocking threads have advanced meanwhile, that node can be safely reclaimed, and the routine return successfully (lines 15-16). The set_block_list() has the goal of resetting the blocking list. In case of failure and before returning, we guarantee that the blocking list remains only with threads that still blocking the node (line 7). This avoids rechecking the threads that have already moved.

The set_list() procedure, shown in Listing 4.7, has the goal of setting the blocking list. When a thread adds the nodes into the reclaim_list, fills the node's blocking list with the threads that are protecting the node from reclamation. In this step, we guarantee that the blocking list is only composed of threads that protect node after their assignment to CTrie, and after his retirement, the threads that start protect node

```
void add_to_reclaim_list(void *entry, int tid){
   size_t hash, int level;
   void* node = UNTAG(entry);
   if (node == NULL)
      return NULL;
   if (IS_SNode(entry)){
      SNode snode = node;
      hash = snode->hash;
      level = snode->level;
      snode->block_list = set_list(hash, level, tid);
   } else if (IS_ANode(entry)){
      ANode *anode = node;
      hash = anode->hash;
      level = anode->level;
      anode->block_list = set_list(hash, level, tid);
   } else if (IS_ENode(entry)){
      ENode *enode = node;
      hash = enode->hash;
      level = enode->level;
      enode->block_list = set_list(hash, level, tid);
   } else if (IS_FNode(entry)){
      FNode *fnode = node;
      hash = fnode->hash;
      level = fnode->level;
      fnode->block_list = set_list(hash, level, tid);
   } else if (IS_CNode(entry)){
      CNode cnode = node;
      hash = cnode->hash;
      level = cnode->level;
      cnode->block_list = set_list(hash, level, tid);
   }
   ReclaimList new_reclaim = alloc_init_reclaim_list(entry, hash, level);
   new_reclaim->next = HA[tid].reclaim_list;
   HA[tid].reclaim_list = new_reclaim;
   HA[tid].reclaim_size++;
}
```

Listing 4.4: Pseudo-code for the add_to_reclam() procedure

will not be considered in the reclamation procedure.

4.5.4 Check for Stucks

This procedure, shown in Listing 4.8, uses two copies of hazard pairs, one taken before and another taken after the corresponding ANode was retired. We check if both copies are equal, meaning that we only want to consider threads that are stopped in the same place (ANode) in the data structure (line 3). Next, we check which copied hazard pairs match with the hazard pair that represents the node being inserted (line 4) and, if they match, we check if the corresponding thread was there before the node was attached or

```
void retire_node(void *entry, int tid){
   void* node = UNTAG(entry);
   void* freeze_entry = PUT_TAG(NULL, FNODE);
   if (IS_ANode(entry)){
      ANode anode = node;
      for (int i=0; i<anode->size; i++){
         void *node_entry = anode->array[i];
         if (CAS(anode->array[i], node_entry, freeze_entry))
            if (UNTAG(node_entry) != NULL)
               reclaim_node(node_entry, tid);   // reclaim nodes recursively
      }
   } else if (IS_SNode(entry)){
      SNode snode = node;
      CAS(snode->pending, NULL, freeze_entry);
   } else if (IS_ENode(entry)){
      ENode enode = node;
      void *node_entry = enode->narrow;
      if (CAS(enode->narrow, node_entry, freeze_entry))
         reclaim_node(node_entry, tid);         // reclaim nodes recursively
   } else if (IS_FNode(entry)){
      FNode fnode = node;
      void *node_entry = fnode->frozen;
      if (CAS(fnode->frozen, node_entry, freeze_entry))
        reclaim_node(oldtg, tid);               // reclaim nodes recursively
   } else if (IS_CNode(entry)){
      CNode cnode = node;
      void *node_entry = cnode->cur;
      if (CAS(cnode->cur, node_entry, freeze_entry))
         reclaim_node(oldtg, tid);              // reclaim nodes recursively
   }
}
```

Listing 4.5: Pseudo-code for the retire_node() procedure

after. For that, we compare the num_op present on the copy and the one in HA (line 5). If both are equal, that means that the thread remained stopped in that position, so we update the stuck array to mark that thread (line 6).

4.5.5 Update Stuck Array

The stuck array is only updated to reflect more recent values, as shown in Listing 4.9. This is guaranteed by checking if the value to be updated has a greater num_op value than the one currently in the stuck array (line 6). Otherwise, the stuck array will be updated to older states that become outdated because the thread is no more in that situation.

The in_stucks() procedure, shown in Listing 4.10, serves the purpose of knowing if a certain thread is in

```
int check_blocking_list(void *node, int tid){
   BlokingList cur_block = get_block_list(node);
   while (cur_block) {
      int block_tid = cur_block->tid;
      size_t num_op = cur_block->num_op;
      int level = cur_block->level;
      BlockingList next_block = cur_block->next;
      if ((num_op == LOCAL_HA[block_tid].num_op) && (tid != block_tid)) {
        set_block_list(node, cur_block);
        return FAILURE;  // in blocking list and same num_op, node is protected
      }
      free(cur_block);
      cur_block = next_block;
   }
   HA[tid].reclaim_size--;
   return SUCCESS;
}
```

Listing 4.6: Pseudo-code for the check_blocking_list() procedure

```
BlockingList set_list(size_t hash, int level, int tid){
   BlockingList header;
   for (int i = 0; i < NUMBER_THREADS; i++) {
     if ((i != tid) && match_hash(level-1, hash, LOCAL_HA[i].hash) &&
         (!in_stucks(LOCAL_HA[i].num_op, level, i)))
      add_to_blocking_list(header, i, LOCAL_HA[i].num_op, level);
   }
   return header;
}
```

Listing 4.7: Pseudo-code for the set_list() procedure

```
void check_for_stucks(size_t hash, int level, int tid) {
   for (int i = 0; i < NUMBER_THREADS; i++) {
      if ((i != tid) && equal_local_hazard_copies(i))
        if (match_hash(level-1, LOCAL_HA2[i].hash, hash)  &&
            (LOCAL_HA2[i].num_op == HA[i].num_op))
            update_stuck_entry(i);
  }
}
```

Listing 4.8: Pseudo-code for the check_for_stucks() procedure

```
void update_stuck_entry(int tid){
   size_t num_op = 0;
   HazardEntry* stuck_entry = STUCK_HA[tid];
   if (stuck_entry)
      num_op = stuck_entry->num_op;
   if (num_op < LOCAL_HA2[tid].num_op) {
      HazardEntry* new_stuck_entry = alloc_init_hazard_entry(LOCAL_HA2[tid]);
      if (CAS(STUCK_HA[tid], stuck_entry, new_stuck_entry))
         free(stuck_entry);
   }
}
```

Listing 4.9: Pseudo-code for the update_stucks() procedure

the stuck array and, consequently, in an outdated state.

```
int in_stucks(size_t num_op, int level, int tid){
  HazardEntry* stuck_entry = STUCK_HA[tid];
  if (stuck_entry)
    if ((stuck_entry->num_op == num_op) && (stuck_entry->level == level))
      return SUCCESS;
  return FAILURE;
}
```

Listing 4.10: Pseudo-code for the in_stucks() procedure

4.5.6 All Together

As we mention before, nodes start by being allocated on memory, and then they pass through the insertion process where they are assigned to the data structure. Next, they are eventually removed from the structure by the remove procedure where they pass through the reclamation process. This life cycle is essential to guarantee some properties and the correct behavior of our solution. In this section, we present again the algorithms that implement this life cycle, as initially presented in section 3.1 but now including the code to support the reclamation process. Remember that now, all these procedures have a new argument that represents the thread id (tid), and the nodes have the new fields described in Listing 4.1. In the case of the lookup_key() and insert_key() procedures, we need to protect the nodes while traversing the CTrie. Listings 4.11 and 4.12 show the changes done to both procedures, which basically update the level being traversing in the hazard array in order to mark nodes that the thread can reach (line 2).

Listing 4.13 shows the modifications to the collision_snode() procedure. In this new version, when a non-empty entry is disassigned from the CTrie (lines 9, 22, 34, 40), the reclaim_node() procedure is now called (lines 10, 25, 37, 41) to perform the reclamation steps of that node. Moreover, before and after the assignment of an ANode entry, a copy of the hazard array HA is done by the copy_ha() procedure and

```
SNode lookup_key(size_t key, size_t hash, int level, ANode cur, int tid){
   update_level(level, tid);
   ANode next;
   ...
}
```

Listing 4.11: Pseudo-code for the lookup_node() procedure

```
int insert_key(size_t key, size_t val, size_t hash, int level, ANode cur, ANode
    prev, int tid) {
   update_level(level, tid);
   int pos = GET_POS(hash, level, cur->size);
   ...
}
```

Listing 4.12: Pseudo-code for the insert_key() procedure

then the stuck array is updated if any thread is in an outdated state, work done by check_for_stuck() procedure.

Listing 4.14 and 4.15 show the changes done to the remove routine. In remove, we also need to protect the nodes before reading them (line 2). Since we can call recursively the remove routine because of the compaction strategy (lines 5-10), we need to update the hazard level at any time we traverse levels (line 7). In the remove_snode procedure, we reclaim nodes after disassigning them (lines 9, 12, 25), and when the algorithm assigns a pending ANode, we need to check_for_stucks, since a new level was assigned to the structure (lines 17-23).

Listing 4.16 and 4.17 show the changes done to the complete_expansion() and complete_compaction() procedures. In both, we start by protecting the corresponding ENode or CNode and the nodes at that level (line 2). Like before, when an ENode is disassigned from the structure, the nodes pass through the reclamation procedure (line 12). The same happens for the CNodes (line 22);

Listing 4.18 shows the changes done to the freeze() procedure. In freeze(), we start by protecting the nodes in that level, since freeze can traverse various levels (line 3). Again, when a pending change is assigned, the old node is reclaimed (line 24). If that pending node is an ANode, a new level will be assigned, so we also update the stuck array (lines 17-23). Since freeze traverses recursively through the levels, we update the level when the routine returns (line 29).

```
int collision_snode(size_t key, size_t val, size_t hash, int level, ANode cur,
    ANode prev, void *entry, int pos, int tid) {
   SNode snode = (SNode)UNTAG(entry);
   void *pending_entry = snode->pending;
   if (UNTAG(pending_entry) == NULL){                          // no pending changes
      if (snode->key == key) {                       // same key, try to update value
         SNode new_snode = alloc_init_snode(key, val, hash, NULL);
         new_snode = PUT_TAG(new_snode, SNODE);
         if (CAS(snode->pending, NULL, new_snode)) {  // annouce pending update
            if (CAS(cur->array[pos], entry, new_snode))        // perform update
               reclaim_node(entry, tid);
            return SUCCESS;
         }
         free(UNTAG(new_snode));
         return insert_key(key, val, hash, level, cur, prev);
      } else if (IS_Narrow(cur))      // different keys and narrow ANode, expand
         expansion(hash, level, cur, prev);
      else {                      // different keys and wide ANode, insert new level
         ANode new_anode = alloc_init_anode_snode(snode->key, snode->val,
             level+1);
         copy_ha(LOCAL_HA, tid);
         if (CAS(snode->pending, NULL, new_anode))  // annouce pending new level
            // perform insertion of new level
            if (CAS(cur->array[pos], entry, new_anode)){
               copy_ha(LOCAL_HA2, tid);
               check_for_stucks(snode->hash, level+1, tid);
               reclaim_node(entry, tid);
            }
         else {
            free(UNTAG(new_anode));
            return insert_key(key, val, hash, level, cur, prev);
      }
   } else {
      else if (IS_ANode(pending_entry){
         copy_ha(LOCAL_HA, tid);
         if (CAS(cur->array[pos], entry, snode->pending)){
            copy_ha(LOCAL_HA2, tid);
            check_for_stucks(snode->hash, level +1, tid);
            reclaim_node(entry, tid);
         }
      } else if (IS_SNode(pending_entry))
         if (CAS(cur->array[pos], entry, snode->pending))
            reclaim_node(entry, tid)
      return insert_key(key, val, hash, level, cur, prev);
   }
   return FAILURE;
}
```

Listing 4.13: Pseudo-code for the collision_snode() procedure

```
int remove_key(size_t key, size_t hash, int level, ANode cur, ANode prev, int
    tid){
   update_level(level, tid);
   int pos = GET_POS(hash, level, cur->size);
   ...
   else if (IS_ANode(entry)) {
      int status = remove_key(key, hash, level+1, (ANode)entry, cur);
      update_level(level, tid);
      if (status && prev != NULL && is_compactable(cur))
         compaction(hash, level, cur, prev);
      return status;
   } ...
}
```

Listing 4.14: Pseudo-code for the remove_key() procedure

```
int remove_snode(size_t key, size_t hash, int level, ANode cur, ANode prev,
    void *entry, int pos, int tid){
   SNode snode = (SNode)UNTAG(entry);
   if (snode->key != key)                      // diferent key, nothing to remove
      return SUCCESS;
   void *pending_entry = snode->pending;
   if (UNTAG(pending_entry) == NULL){          // same key and no pending changes
      if (CAS(cur->array[pos], entry, NULL)) {
         if (IS_FNode(pending_entry)){             // node froozen in the meantime
            reclaim_node(entry, tid);
            return FAILURE;
         }
         reclaim_node(entry, tid);
         return SUCCESS;
      }
      return remove_key(key, hash, level, cur, prev);
   } else if (!IS_FNode(pending_entry))                        // pending change
      if (IS_ANode(pending_entry))
         copy_ha(LOCAL_HA, tid);
      // perform pending change
      if (CAS(cur->array[pos], entry, snode->pending)){
         if (IS_ANode(pending_entry)){
            copy_ha(LOCAL_HA2, tid);
            check_for_stucks(key, level+1, tid);
         }
         reclaim_node(entry, tid);
      }
   return FAILURE;
}
```

Listing 4.15: Pseudo-code for the remove_snode() procedure

```
void complete_expansion(ENode enode, int tid){
   update_level(enode->level, tid);
   freeze(enode->narrow);
   ANode wide = alloc_init_wide_anode(enode->narrow, enode->level);
   if (!CAS(enode->wide, NULL, wide)) {          // if wide already assigned
      free(wide);
      wide = enode->wide;                                      //read wide
   }
   int prev_pos = enode->prev_pos;
   void* enode_entry = PUT_TAG(enode, ENODE);
   if (CAS(enode->prev->array[prev_pos], enode_entry, wide))
      reclaim_node(enode_entry, tid);
}
```

Listing 4.16: Pseudo-code for the complete_expansion() procedure

```
int complete_compaction(CNode cnode, int tid){
   update_level(cnode->level, tid);
   ANode anode = (ANode)cnode->cur;
   freeze(anode);
   void *compact = NULL;
   for (int i=0; i < anode->size; i++) {
      void *node = anode->array[i];
      if (IS_SNODE(node) && (compact == NULL))
         compact = node;
      else if (UNTAG(node) != NULL){
         compact = anode;
         break;
      }
   }
   if (IS_SNode(compact)) {                        // only one SNode found
      compact = alloc_duplicate_snode(compact);
   } else if (IS_ANode(compact) { // at least two SNodes or another node found
      compact = alloc_duplicate_anode(compact);
   }
   void *cnode_entry = PUT_TAG(cnode, CNODE);
   if (CAS(cnode->prev->array[cnode->prev_pos], cnode_entry, compact))
      reclaim_node(cnode_entry, tid);
   return ((compact == NULL) || IS_SNode(compact));
}
```

Listing 4.17: Pseudo-code for the complete_compaction() procedure

```
void freeze(ANode cur, int tid){
   int i = 0;
   update_level(cur->level, tid);
   while (i < cur->size) {
      void *entry = cur->array[i];
      void *node = UNTAG(entry);
      if (entry == NULL)                          // if NULL, tag it as frozen
         if (CAS(cur->array[i], entry, PUT_TAG(NULL, FNODE)))
            i++;
      else if (IS_SNode(entry)){
         SNode snode =(SNode)node;
         void *pending_entry = snode->pending;
         void *pending = UNTAG(pending_entry);
         if (pending_entry == NULL)  // SNode without pending changes, tag
             pending field
            if (CAS(snode->pending, NULL, PUT_TAG(NULL,FNODE)))
               i++;
         else if (!IS_FNode(pending_entry)){                  // pending change
            copy_ha(LOCAL_HA, tid);
            if (CAS(cur->array[i], entry, snode->pending)){ // perform pending
                change
               if(IS_ANODE(pending_entry)){
                  copy_ha(LOCAL_HA2, tid);
                  check_for_stucks(snode->hash, snode->level+1, tid);
               }
               reclaim_node(entry, tid);
            }
         }
      }  else if (IS_FNode(entry) && (node !=NULL)){
         freeze((FNode)node->frozen);
         update_level(level, tid);
      }  else if (IS_ENode(entry))
         complete_expansion(node);
      else if (IS_CNode(entry))
         complete_compaction(node);
   }
}
```

Listing 4.18: Pseudo-code for the freeze() procedure

Chapter 5

Experimental Results

In this chapter, we explain how benchmarking was done. We also present the experimental results of such benchmarking over the different versions and discuss the results.

5.1 Methodology

We have implemented various versions of the API to support and manage the data structure. To make use of the API, we used the benchmark tool proposed by Moreno et al. [2019]. This tool compiles the version of the API that will be used, wrapped with a module responsible to control the execution environment. Our benchmark tool receives as input 6 parameters:

- The number T of threads to be used;
- The number N of operations to be performed;
- The percentage Pi of N that correspond to insert operations;
- The percentage Pr of N that correspond to remove operations;
- The percentage Psf of N that correspond to search found operations;
- The percentage $Psnf$ of N that correspond to search not found operations.

Figure 5.1 represents the benchmark tool, in which the controller receives the 6 parameters and communicates with the CTries through the specific API. The benchmark tool can be divided into different stages. In the beginning, the benchmark tool prepares the execution environment, this step depends on the API version that will be executed. For example, our proposal starts by creating the CTrie root reference, the Hazard array, and the Stuck array. Next, the tool runs the given number of threads T to insert beforehand on data structure the values that will be searched or removed. Each thread is responsible to perform the same number of operations, this way, the number of operations to perform (N) should be divisible by the number of threads (T). Threads also had a pre-defined seed to use in the pseudo-random number generator (PRNG). This PRNG is responsible for the keys generation. The key range is divided in a way that matches the percentages of operations. This means that a key generated by PRNG also includes the operation that will be performed. In the next stage, the tool executes the benchmarks itself. The thread's seeds are reset to the pre-defined values and start counting the execution

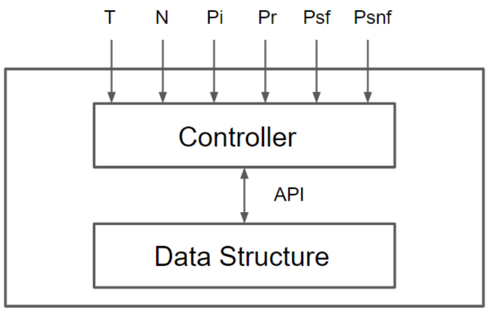

Figure 5.1: Benchmark Tool Diagram

time. Like in the previous stage, each thread executes the operations corresponding to the generated PRNG values. The final step of this stage is to present the execution time. After all, threads have completed their operations. There is also a third stage that performs inspections on the data structure and alerts if something is not accordingly to what is supposed to be. This stage is optional since it works like a debugging tool since the benchmark itself was already done. Like in the previous stages, the tool resets the seed array to the pre-defined values. In this final stage, the threads only perform searching operations. Such searches will give us the information if the operations done in the previous stage were correctly done. If a thread performs a search over a key that belongs to the remove range, this search should return a not found result. Otherwise, if the key is in the insert range, the search should return the respective value. This step provides a strong way to test if the API is working properly. Figure 5.2 represents how keys are divided into the threads and the respective operations.

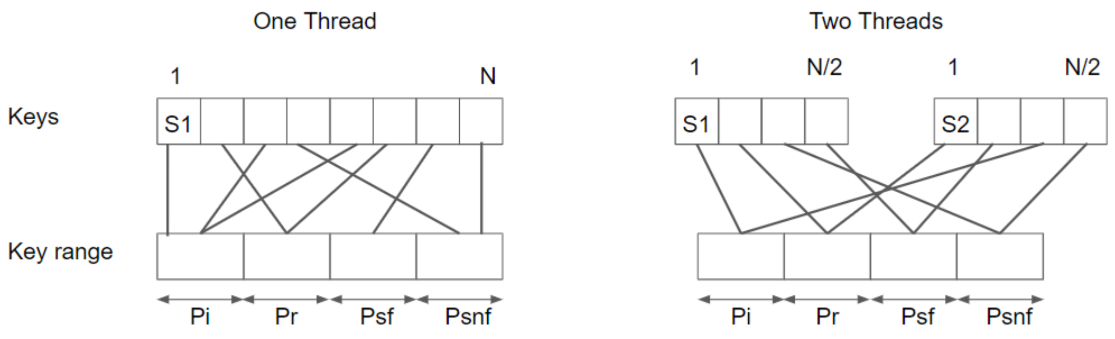

Figure 5.2: Key Distribution into threads and operations

The pseudo-random number generator (PRNG) used is the nrand48_r from the C standard library. This function uses the linear congruential algorithm that is represented by Eq. 5.1. The value of a is 25214903917, the value of c is 11 and the value of m is 2^{48}.

$$f(n+1) = (a \times f(n) + c) \mod m \tag{5.1}$$

Because of the natural property of the randomly generated values, the percentages Pi, Pr, Psf, and Psnf of operations to perform may not be precisely hit in some benchmark runs. However, as the nrand48_r function presents good properties and the number of operations is large enough, the deviation can be considered negligible. In these experiments, we have implemented the following versions of the CTries data structure:

- *NF* (No Free): Version based on Prokopec [2018] and Prokopec et al. [2012] , like explained in section 3.1, where no memory is reclaimed;
- *OF* (Optimistic Free): As explained in section 2.4.1, this version takes an optimistic approach, where each thread has a reclamation ring buffer that holds the nodes to be reclaimed. At the time of reclaiming a node, threads go around the buffer and reclaim the node in that position before assigning the newly retired one. This is theoretically wrong but can be implemented in practice since the buffer is big enough in a way that nodes that are being reclaimed are too old to be referred by any thread. This version provides a great baseline for comparison purposes since it achieves the best performance of all memory reclamation schemes;
- *HE* (Hazard Eras): This version is an implementation of the idea presented in section 2.4.5, on top of our base version of the CTries. It uses a global clock and an array of thread's local clocks. When nodes are assigned to the data structure, they save the current global era (insertion time). Threads can reclaim nodes whose insertion time is greater than the threads eras. When a thread reads a new reference, it updates its local clock with the value of the global one. Since CTries produce memory garbage in the insert operations, the global clock should be incremented not just on the remove operations but also in the insert operations. This way, the threads that stop before the node was created, do not block the node from being reclaimed.
- *IBR* (Interval Based Reclamation): Section 2.4.6 explain the different approaches to implement this method, we used the persistent data structure version because it is the most compatible approach to our data structure. This method is based on the *HE* method but instead of just reserving all the unretired nodes before a given era, it reserves only the nodes whose lifetimes eras were reserved. This approach saves on the nodes the inserted era (the global era when the node is inserted) and the remove era (the global era when the node is removed). This way, if there are no threads in that lifetime, nodes can be reclaimed.
- *HHL* (Hazard Hash and Level): This version is the direct implementation of the method proposed by Moreno et al. [2019] with some little differences due to the data structure, as explained in section 4.2. This method has a global hazard array which marks the paths the threads are traversing. This approach does not provide bounded memory in our case, but it is interesting to note the overhead compared to our adaption (HHL_BS) to achieve bounded memory. It is also a good approach to compare the LFHT HHL with the CTries HHL since the same machine and benchmark tool was used to run the experiments.
- *HHL_BS* (Hazard Hash and Level with Blocking list and Stuck array): This version represents the implementation of our solution, as proposed in chapter 4 , that provides bounded memory usage. This method uses a blocking list on the nodes and a global Stuck array, to transform the HHL solution into a bounded one. Due to the lack of time, some aspects of this solution are not considered, like the case of hash collision at the last level of the structure.

All these versions were tested with a reclamation threshold of 256 nodes and different percentages of operations. To compare the results obtained in our experiments, we use the benchmarks proposed by Moreno et al. [2019]. Experiments were executed with a hash function equivalent to the identity (h(x)=x), in order to minimize possible differentiating factors and thus reduce the overall overhead. Note that the keys are already random and the way the key space is divided already prevents interferences. All experiments were run 5 times and the results presented are the average of such runs.

5.2 Results and Discussion

The machine used to run our experiments was a NUMA with two AMD Opteron Processor 6274 and 32GiB of ECC RAM. The memory allocator used was jemalloc Evans [2006] version 5.0 as it showed good results and was able to scale with all the cores used without generating contention in the kernel. However, ideally, we would like to use a lock-free memory allocator. The experiments showed next use a fixed size of 10^6 operations but with a varying percentage of insert, remove, and search operations. Appendix A shows the execution time results for all the experiments done.

5.2.1 Baseline

Figures 5.3 to 5.7, present the comparison between the *NF* and *OF* versions. In terms of the execution time, in seconds, required to execute the 10^6 operations on 5 different benchmarks. The insert and the remove operations are the ones in which the reclamation procedure is triggered, and this is clear in Figs. 5.3, 5.4 and 5.6, which show the major impact of the reclamation procedure. The searching operations remain almost without overhead since lookups do not call the reclamation procedure, this is shown in Fig. 5.5. Figure 5.7 shows the impact of the reclamation in a more general run that somehow tries to simulate a closer approach to a real application of the data structure, with all the operations represented. The overhead of the memory allocator is almost minimal because of the use of the thread's local caches implemented in modern memory allocators like jemalloc. These local caches permit that the allocator presents minimal synchronization between the threads since allocations and deallocations remain equals and interleaved enough.

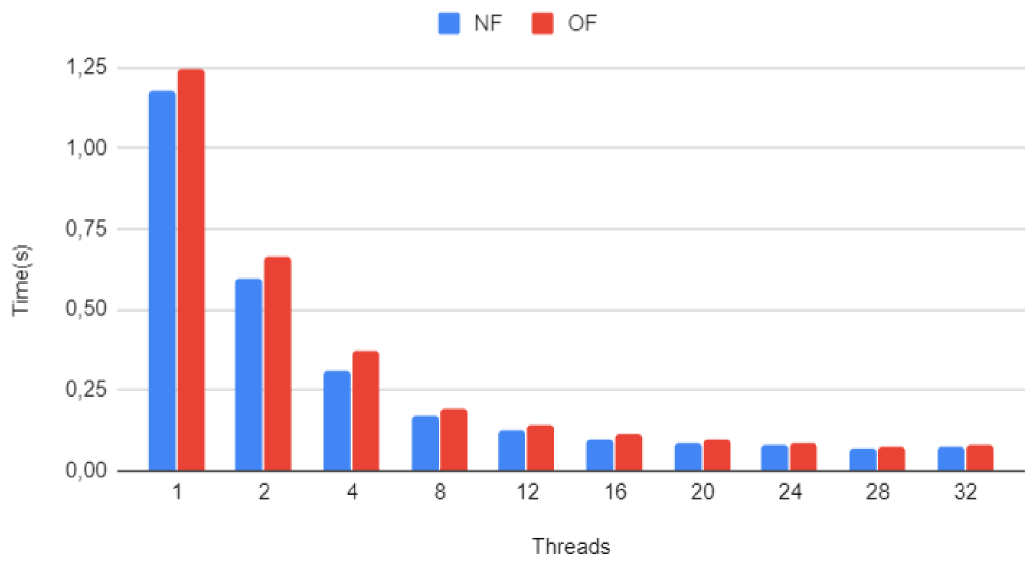

Figure 5.3: Execution time, in seconds, for a dataset with 100% inserts (10^6 operations in total) for the NF and OF versions

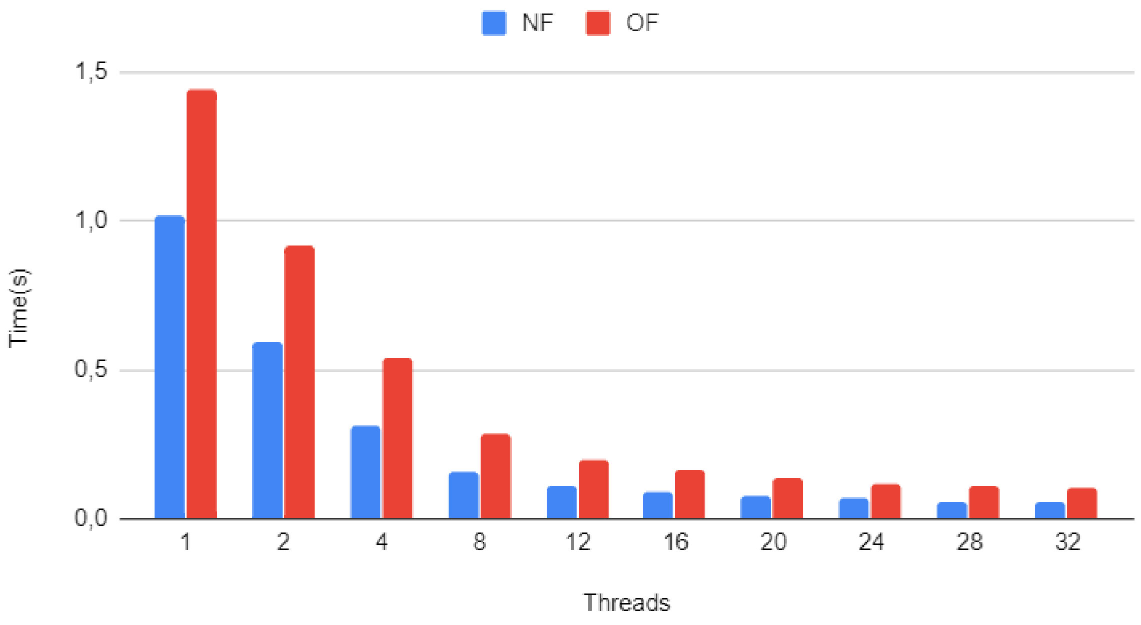

Figure 5.4: Execution time, in seconds, for a dataset with 100% removes (10^6 operations in total) for the NF and OF versions

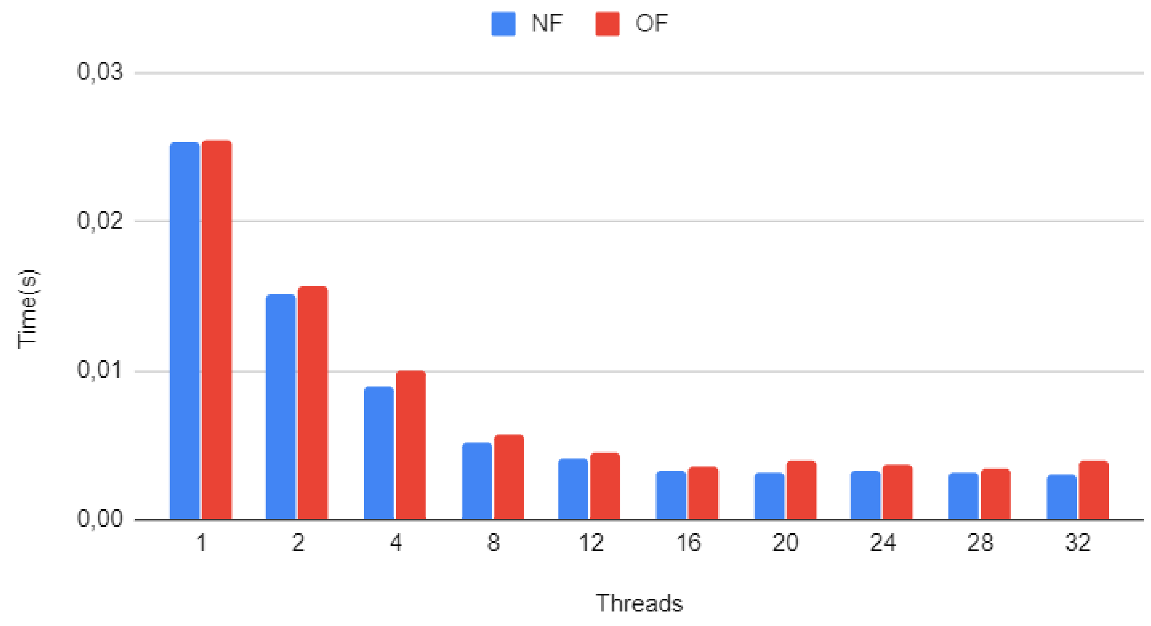

Figure 5.5: Execution time, in seconds, for a dataset with 50% found and 50% not found searches (10^6 operations in total) for the NF and OF versions

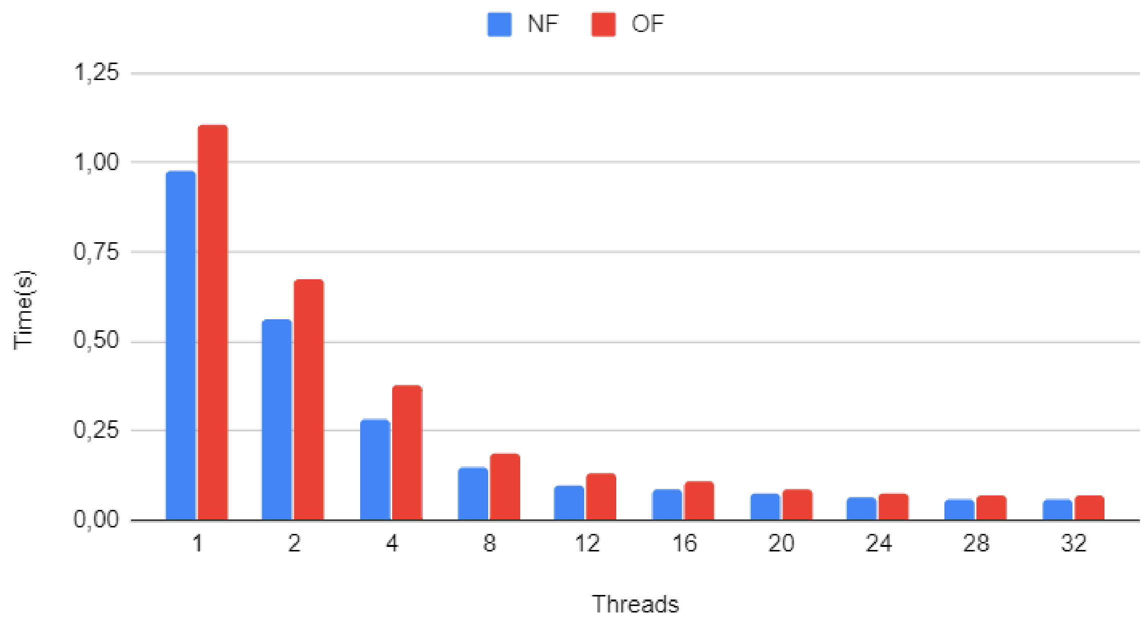

Figure 5.6: Execution time, in seconds, for a dataset with 50% inserts and 50% removes (10^6 operations in total) for the NF and OF versions

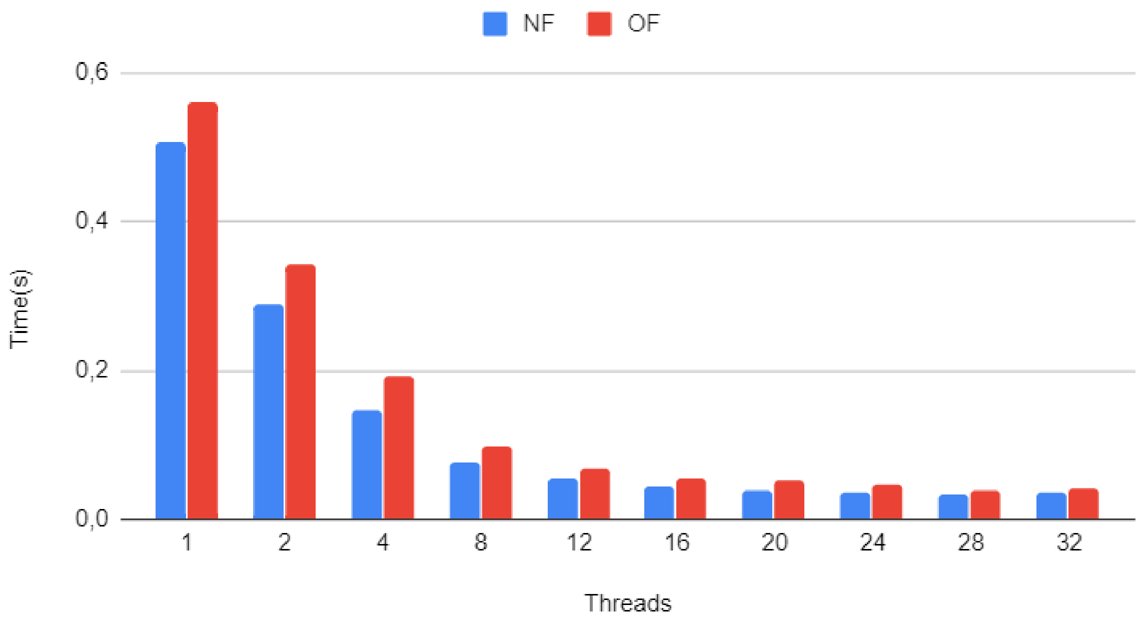

Figure 5.7: Execution time, in seconds, for a dataset with 25% inserts, 25% removes, 25% found and 25% not found searches (10^6 operations in total) for the NF and OF versions

5.2.2 Memory Reclamation Impact

Now, we compare our baseline *OF* with the different reclamation methods implemented (*HE*, *IBR*, *HP*, *HHL* and *HHL_BS*). To better compare the overhead of each memory reclamation method, all results are normalized to our baseline (OF). Next, Figs. 5.8 to 5.12 compare the results of the different memory reclamation methods on the same 5 benchmarks used before.

Figure 5.8 represents the benchmark with 100% insert operations. In inserts operations, *HHL_BS* and *IBR* show heavy degradation because these methods need additional steps not just on the reclamation scenario but also when inserting nodes. The *HHL_BS* has the most overhead due to the Stuck array management and the double read of the hazard array each time a level is inserted. In fact, the insert procedure should be the main focus to improve once the double read of the hazard array and the management of the Stuck array is the main cause of the overhead of our solution. The insertion time of *IBR* has some overhead in insertion too due to the management of the global clock and, when a node is created, the global clock needs to be read in order to set it on the node.

The *HHL* behavior is very close to ideal, however, it is important to remember that this solution does not provide bounded memory. The *HE* and the *HP* have similar results, they both have overhead just on the reclamation scenario and do not have any special care when inserting nodes.

In the case of remove operations, the *HHL_BS* and the *IBR*, recover some of the time spent on the insertion of nodes. Figure 5.9 demonstrates that both methods present better performance than the *HE* and the *HP*. The *HE* has heavy degradation in this benchmark because the reclamation list of threads grows very fast, and the traversal of that list consumes some of the time in the reclamation procedure. This happens because, since all threads are just doing removes, they are inside the lifetimes of the nodes in the structure. Meanwhile, as the threads update the Era, the nodes start being reclaimed. This is one of the problems of the large bounded memory that permits that threads have large reclamation lists.

In the search operations, represented by Fig. 5.10, we can notice a greater degradation of the performance

in all methods, comparing with our baseline. This happens because the *OF* does not need any kind of additional steps in the search operations, while the other methods need to protect the nodes being traversed, therefore adding overhead to all the operations and not just the ones that reclaim nodes.
Figure 5.11 that represents 50% of inserts and 50% of removes, we can see how the different approaches become closer to each other, with the exception of the *IBR*, which we believe can be related with the bad choice of the internal thresholds. As this method and the *HE* should present similar results.
Figure 5.12 represents the general scenario with all operations. In overall, the results show that the *HHL* is the most closely to ideal approach, however, our bounded solution *HHL_BS* shows overheads that make it similar to the other bounded reclamation methods. In these experimental results, all versions use the same thresholds, but they provide different bounded limits. If thresholds had been adapted to approximate the different bounds, maybe the experimental results will be more closely to the baseline.

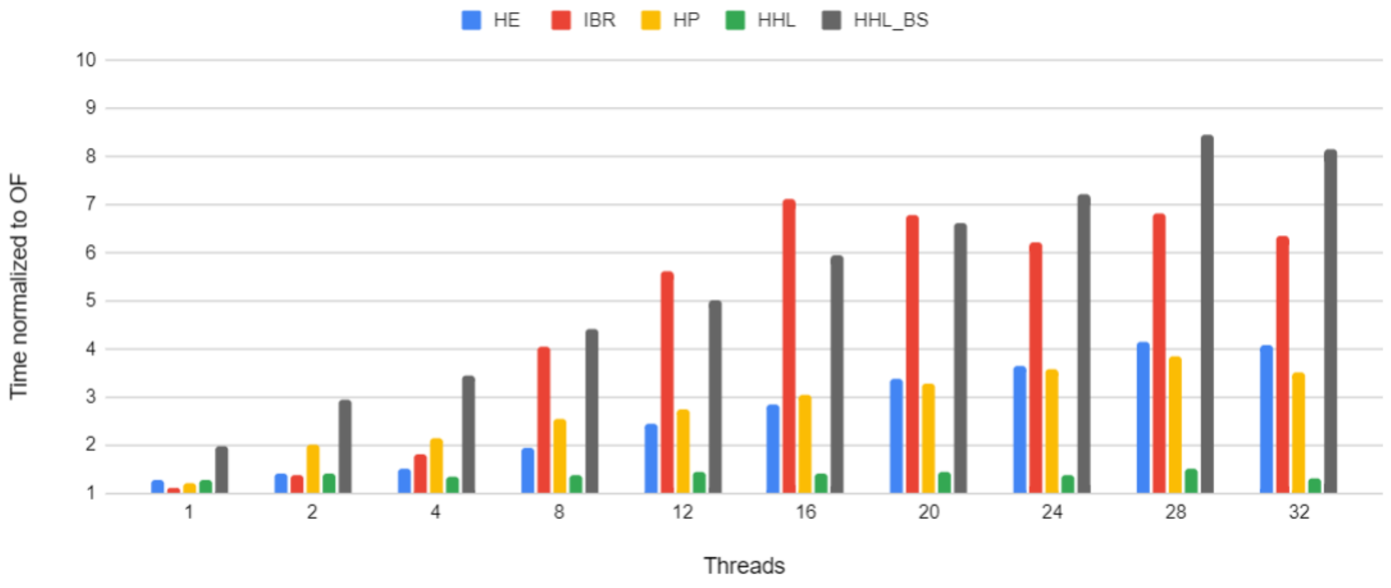

Figure 5.8: Execution time, normalized to OF, for a dataset with 100% inserts (10^6 operations in total) for the MR methods

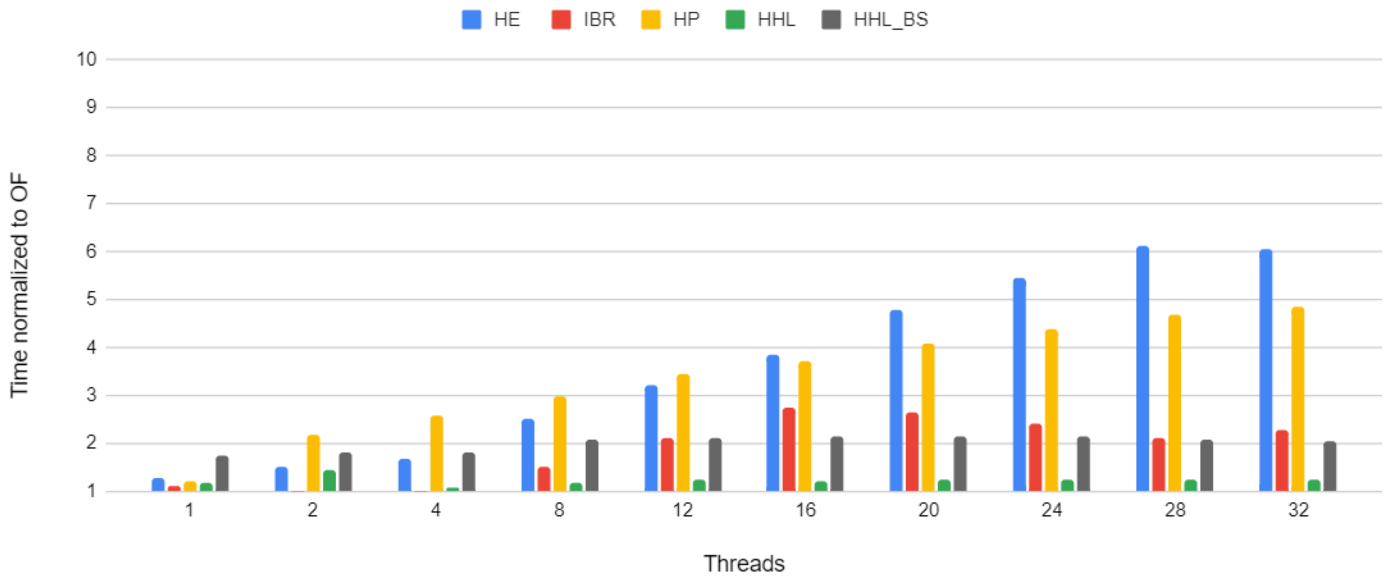

Figure 5.9: Execution time, normalized to OF, for a dataset with 100% removes (10^6 operations in total) for the MR methods

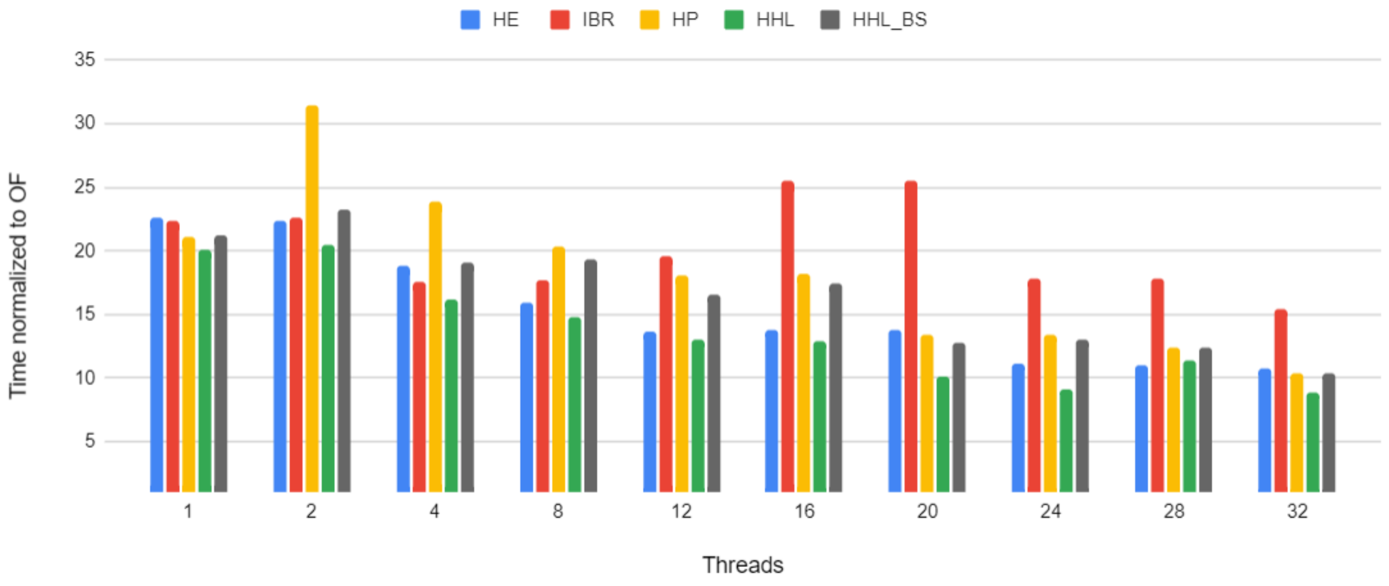

Figure 5.10: Execution time, normalized to OF, for a dataset with 50% found and 50% not found searches (10^6 operations in total) for the MR methods

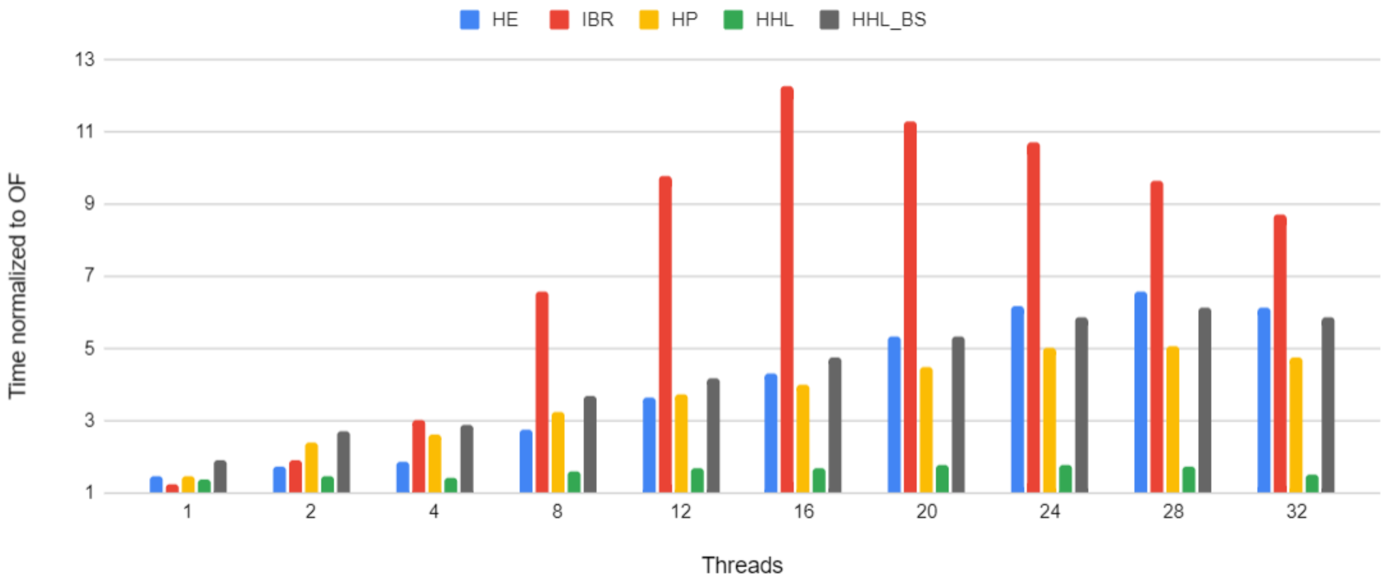

Figure 5.11: Execution time, normalized to OF, for a dataset with 50% inserts and 50% removes (10^6 operations in total) for the MR methods

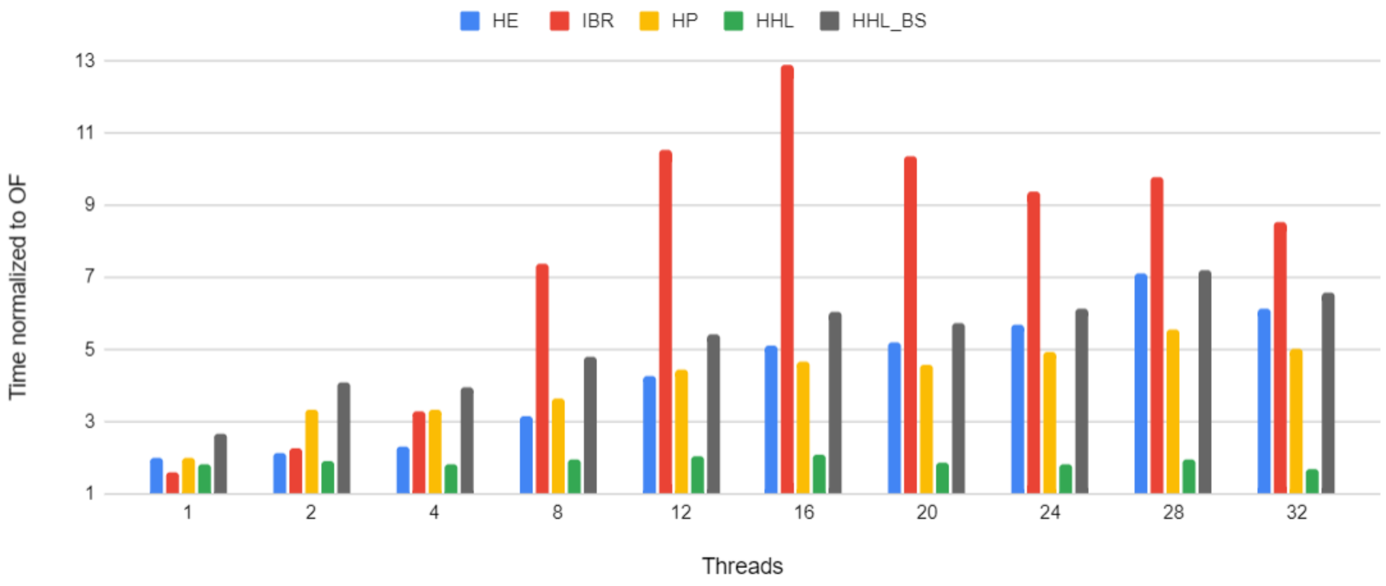

Figure 5.12: Execution time, normalized to OF, for a dataset with 25% inserts, 25% removes, 25% found and 25% not found searches (10^6 operations in total) for the MR methods

5.2.3 Comparison with LFHT

Next, we compare our runs with the *HHL* results obtained by Moreno et al. [2019] for the LFHT data structure. In Fig. 5.13 are represented the runs only with searching operations. Both data structures show similar performance for the searching operations, but the *LFHT* shows slightly better results. This might happen due to the need for the hazard pairs in *CTries* to be updated on every level. *LFHT* reduces this overhead since it only updates the level when reaching a leaf node. Overall, the *LFHT* presents significantly better performances, however, it is important to note that both structures have different properties. The *CTries* reclaims all different types of nodes and, most often, more than one node at a time, bringing overheads that are not presented in the LFHT data structure. This is shown in Figs. 5.14 and 5.15.

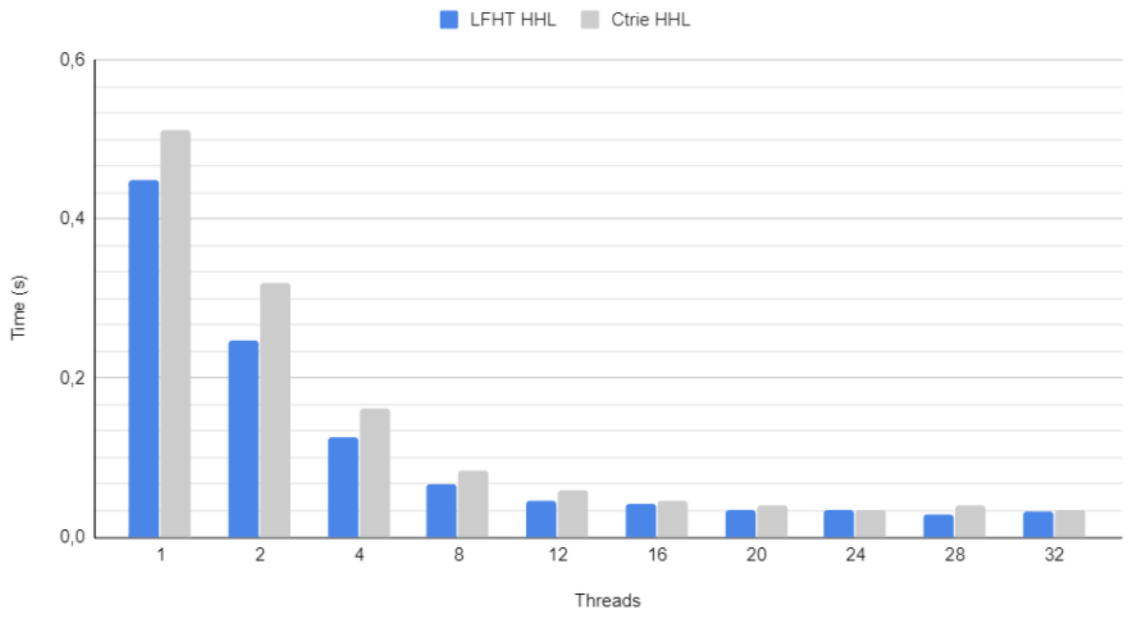

Figure 5.13: Execution time, in seconds, for a dataset with 100% searches (10^6 operations in total) for the LFHT and CTries versions using HHL

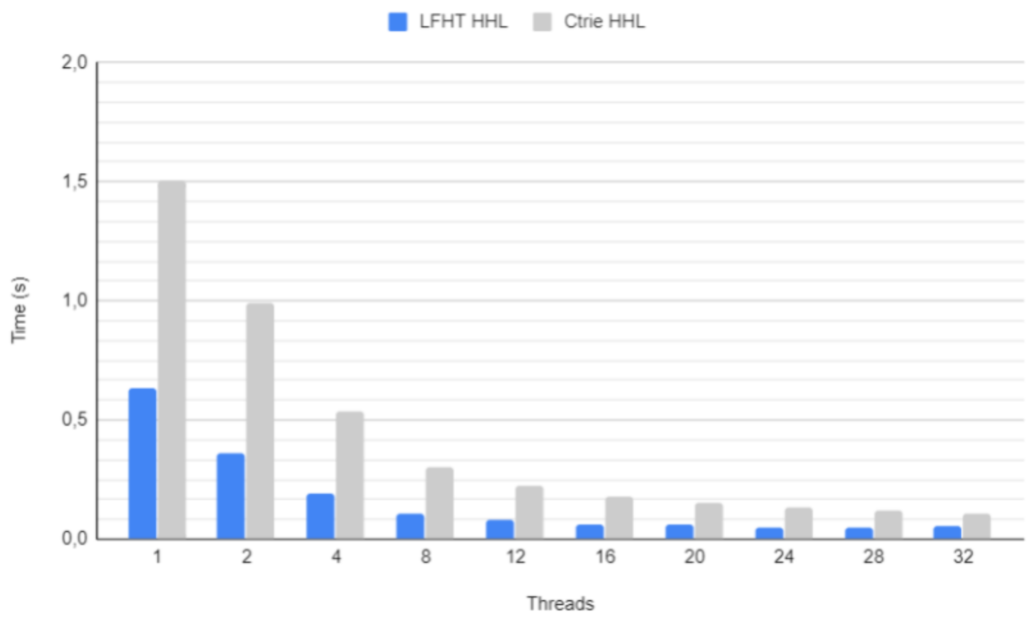

Figure 5.14: Execution time, in seconds, for a dataset with 50% inserts and 50% removes (10^6 operations in total) for the LFHT and CTries versions using HHL

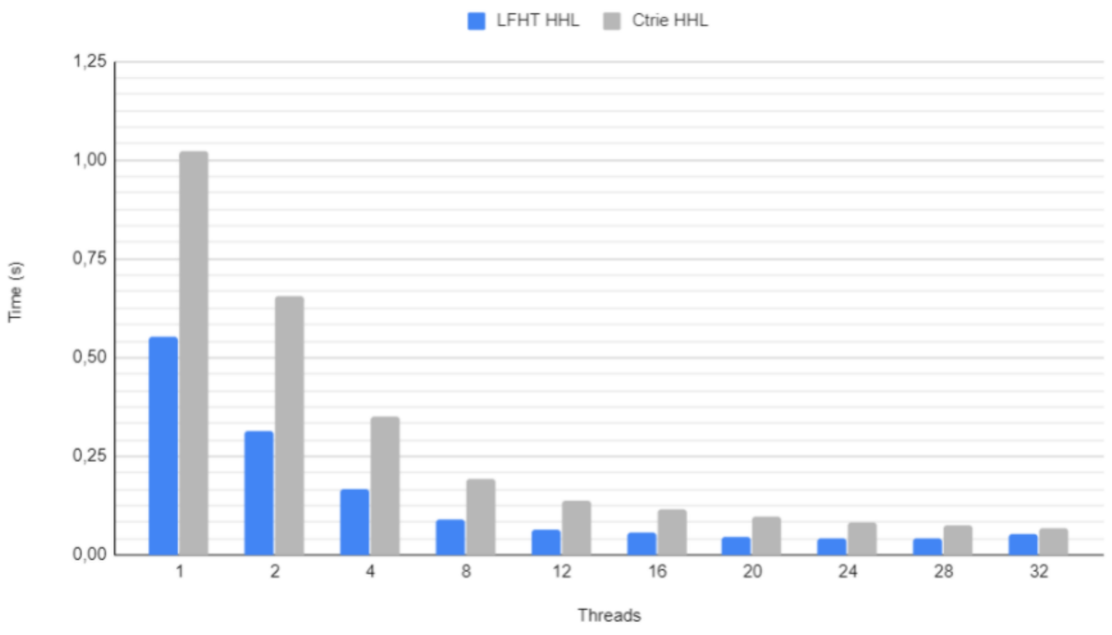

Figure 5.15: Execution time, in seconds, for a dataset with 25% inserts, 25% removes and 50% searches (10^6 operations in total) for the LFHT and CTries versions using HHL

Chapter 6

Conclusion

In this chapter, we resume our work, the main contributions provided by the thesis and we propose further research directions. This thesis started with the purpose of reclaiming memory from the CTries data structure, providing the possibility of a truly lock-free implementation outside garbage collection environments. We started by studying the current state-of-the-art memory reclamation methodologies and the different versions of the CTries. We also studied the LFHT data structure as a way to better understand the relationship between memory reclamation methods (and in specific the *HHL* method) and data structures. After that initial knowledge about memory reclamation methods and lock-free data structures, we understood that, due to CTries properties, some changes need to be done in order to implement a reclamation method on top of the base implementation. Otherwise, the ABA problem and the persistent pointers problem could happen and lead the data structure to incorrect behaviors. At this point, we redesign some aspects of the data structure to guarantee that nodes follow the correct reclamation cycle and support different implementations of memory reclamation methods. The implementation of *HHL* could also be done, however, we notice that it does not fit our goals since it does not guarantee memory bounded usage in this context. In order to guarantee bounded memory, an adaption of *HHL* was designed and implemented. Comparing both versions of *HHL*, it is clear that our adaption comes with some extra overhead and performance degradation. This can be explained with the procedures that run at the time of creating and removing nodes, or because of not optimal threshold interval. Our experimental results show that *HHL* is the most efficient method in our data structure, however, our approach to making it bounded approximate this method with other state-of-the-art methods. In general, the *HHL_BS* solution provides a bounded adaption of the *HHL* that is very competitive with the other state-of-the-art methods. We expect that the work done in this thesis will inspire others in order to improve other solutions in this area. Further work can include the following tasks:

More experiments. Further experiments would promote a deeper analysis of the memory usage of CTries. The memory reclamation method can be optimized to be better integrated with CTries. For example, the insert procedure should be the main focus to improve as the double read of hazard array and the management of the Stuck array are an important cause of overhead. One idea should be to only update the Stuck array when some threshold is passed, and not every time a level is inserted.

Different approaches to make HHL bounded. As we saw, our adaption of *HHL* introduces overheads in order to guarantee bounded memory usage. Different ideas and approaches can achieve different overheads, and possibly better designs can be achieved.

Adding CTries features. There are various versions of CTries, and several extra features can be integrated, which will require specific integration with the memory reclamation methods. Specific solutions can be designed to fit with those extra features and thus provide special characteristics to each version of the data structure.

Extension to similar data structures A different direction is to try to apply our proposes to similar data structures, mainly data structures with a tree based hierarchy. A more general goal is to try to apply it to other different data structures.

Appendix A

Tables

This appendix serves to show execution time in seconds for all tested configuration. The execution times shown are the average of five runs.

Threads	NF	OF	HE	IBR	HP	HHL	HHL_BS
1	1.179947	1.246026	1,574332	1,389182	1,497708	1,610107	2,478408
2	0.595134	0.661269	0,931164	0,920878	1,323149	0,943471	1,944268
4	0.308740	0.370016	0,560513	0,672386	0,79689	0,494294	1,278514
8	0.169650	0.195026	0,382554	0,791959	0,498448	0,271381	0,859786
12	0.125330	0.142640	0,347023	0,800658	0,391836	0,207141	0,714524
16	0.100467	0.113492	0,323992	0,80787	0,345535	0,162084	0,677074
20	0.087116	0.097352	0,329697	0,661433	0,319452	0,140749	0,644036
24	0.079503	0.089106	0,325553	0,555147	0,319251	0,122329	0,642913
28	0.070740	0.075207	0,311021	0,512366	0,289133	0,112542	0,636168
32	0.074712	0.079085	0,322162	0,501802	0,278853	0,104588	0,645142

Table A.1: 10^6 operations, 100% inserts, 0% removes, 0% searches found, 0% searches not found

Threads	NF	OF	HE	IBR	HP	HHL	HHL_BS
1	1,01942	1,44432	1,863192	1,623745	1,747102	1,714518	2,523039
2	0,597755	0,914247	1,375272	0,855117	1,988008	1,316728	1,650772
4	0,309407	0,539304	0,908286	0,489852	1,384402	0,587165	0,979394
8	0,160076	0,286751	0,718293	0,430924	0,852909	0,337157	0,598082
12	0,109543	0,199567	0,641665	0,418557	0,685013	0,248002	0,423082
16	0,092026	0,164973	0,637728	0,450699	0,614425	0,200798	0,356845
20	0,076208	0,137769	0,656976	0,366525	0,562232	0,172306	0,297688
24	0,069365	0,120454	0,658319	0,28978	0,529908	0,150107	0,258392
28	0,059068	0,109101	0,668425	0,229929	0,509606	0,136058	0,228267
32	0,056663	0,100488	0,608779	0,227961	0,486839	0,125151	0,205014

Table A.2: 10^6 operations, 0% inserts, 100% removes, 0% searches found, 0% searches not found

Threads	NF	OF	HE	IBR	HP	HHL	HHL_BS
1	0,025355	0,025504	0,577404	0,571528	0,538662	0,51221	0,540607
2	0,01518	0,015686	0,350756	0,353807	0,493073	0,320518	0,363626
4	0,008989	0,010024	0,188169	0,176203	0,239563	0,161979	0,190831
8	0,005124	0,005701	0,091106	0,101067	0,116199	0,084176	0,11041
12	0,004056	0,004509	0,06183	0,088422	0,081339	0,058946	0,074612
16	0,003244	0,003508	0,048253	0,089719	0,063664	0,045216	0,061263
20	0,003098	0,003962	0,041657	0,074961	0,052976	0,040105	0,050561
24	0,003327	0,003693	0,041105	0,065647	0,049551	0,033435	0,048279
28	0,003191	0,003442	0,03809	0,061454	0,042861	0,039303	0,042844
32	0,003057	0,003917	0,042325	0,060672	0,040497	0,034923	0,040558

Table A.3: 10^6 operations, 0% inserts, 0% removes, 50% searches found, 50% searches not found

Threads	NF	OF	HE	IBR	HP	HHL	HHL_BS
1	0,979464	1,10811	1,616719	1,378471	1,625252	1,505195	2,109905
2	0,563698	0,67226	1,174917	1,295736	1,606053	0,989785	1,830908
4	0,282967	0,37682	0,700362	1,143773	0,994426	0,536256	1,092953
8	0,149428	0,18962	0,518031	1,249326	0,613909	0,305245	0,699929
12	0,099514	0,131907	0,482986	1,291899	0,489888	0,222273	0,552669
16	0,085831	0,106289	0,456552	1,30381	0,424576	0,180473	0,504164
20	0,073375	0,08739	0,46614	0,986503	0,391364	0,152962	0,466326
24	0,062827	0,075312	0,466825	0,806655	0,378067	0,132268	0,443196
28	0,059803	0,070357	0,461819	0,679102	0,356549	0,120556	0,431449
32	0,060898	0,072484	0,444129	0,632555	0,343243	0,109467	0,426096

Table A.4: 10^6 operations, 50% inserts, 50% removes, 0% searches found, 0% searches not found

Threads	NF	OF	HE	IBR	HP	HHL	HHL_BS
1	0,507283	0,561739	1,117574	0,906577	1,127342	1,023411	1,506965
2	0,288584	0,342363	0,723766	0,772438	1,139357	0,656463	1,395611
4	0,145661	0,192172	0,444908	0,63488	0,636743	0,348993	0,756392
8	0,075624	0,099098	0,313913	0,731477	0,361936	0,194579	0,473533
12	0,054053	0,067417	0,288859	0,710946	0,298505	0,138787	0,365078
16	0,044422	0,054698	0,279754	0,705547	0,255728	0,114718	0,330131
20	0,039409	0,051586	0,268257	0,5355	0,236392	0,095713	0,296723
24	0,035681	0,046696	0,266284	0,437523	0,230921	0,083854	0,286854
28	0,032899	0,038329	0,272719	0,374377	0,213657	0,075247	0,276719
32	0,035821	0,040982	0,252037	0,350099	0,205614	0,068299	0,270334

Table A.5: 10^6 operations, 25% inserts, 25% removes, 25% searches found, 25% searches not found

Bibliography

Miguel Areias and Ricardo Rocha. A lock-free hash trie design for concurrent tabled logic programs. *International Journal of Parallel Programming*, 44(3):386–406, 2016.

Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–24, 2018.

Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*, 2006.

Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2011a.

Maurice Herlihy and Nir Shavit. On the nature of progress. In *International Conference On Principles Of Distributed Systems*, pages 313–328. Springer, 2011b.

HT Kung and Philip L Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems (TODS)*, 5(3):354–382, 1980.

Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

Pedro Moreno, Miguel Areias, and Ricardo Rocha. Memory reclamation methods for lock-free hash tries. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 188–195. IEEE, 2019.

Aleksandar Prokopec. Cache-tries: concurrent lock-free hash tries with constant-time operations. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–151, 2018.

Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.

Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.