A Sleek Lock-Free Hash Map in an ERA of Safe Memory Reclamation Methods

Pedro Moreno[a], Miguel Areias[b], Ricardo Rocha[b]

[a]*Instituto de Astrofísica e Ciências do Espaço, Faculty of Sciences, UPorto, Porto, Portugal*
[b]*CRACS/INESC TEC, Faculty of Sciences, UPorto, Porto, Portugal*
Email: {pmoreno,miguel-areias, ricroc}@dcc.fc.up.pt

Abstract

Lock-free data structures have become increasingly significant due to their algorithmic advantages in multi-core cache-based architectures. Safe Memory Reclamation (SMR) is a technique used in concurrent programming to ensure that memory can be safely reclaimed without causing data corruption, dangling pointers, or access to freed memory. The ERA theorem states that any SMR method for concurrent data structures can only provide at most two of the three main desirable properties: Ease of use, Robustness, and Applicability. This fundamental trade-off influences the design of efficient lock-free data structures at an early stage. This work redesigns a previous lock-free hash map to fully exploit the properties of the ERA theorem and to leverage the characteristics of multi-core cache-based architectures by minimizing the number of cache misses, which are a significant bottleneck in multi-core environments. Experimental results show that our design outperforms the previous design, which was already quite competitive when compared against the Concurrent Hash Map design of the Intel's TBB library.

Keywords: Concurrent Data Structures, Lock-Freedom, Safe Memory Reclamation, ERA Theorem

1. Introduction

The traditional approach to synchronize access to critical sections is to use blocking primitives, such as, mutexes or semaphores. An algorithm is non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. In general, non-blocking algorithms use atomic read-modify-write primitives, the most notable of which is the CAS (compare-and-swap) operation. A non-blocking algorithm is also lock-free if there is guaranteed system-wide progress, i.e., there is no per-thread progress guarantee (individual threads can starve) but it is guaranteed that at least one thread progresses in some well-defined number of steps, regardless of the scheduling policy. Lock-freedom is an important property that is known to offer significant advantages in terms of algorithm design, performance and scalability, therefore improving the overall throughput of concurrent data structures.

Data structures, as an essential building block for most algorithms and systems, are highly desirable to have the lock-freedom property. However, for data structures to be truly lock-free, they require additional memory management methods, usually known as *Safe Memory Reclamation (SMR)* methods, in order to reuse memory from removed *nodes* (i.e., the fundamental elements of a data structure). Memory reclamation on lock-free data structures is a much harder problem to solve, compared to their lock-based counterparts, since exclusive access to any region of the data structure can not be expected without violating the lock-free properties. Even though SMR plays a crucial role in ensuring that the system as a whole is lock-free and can also significantly impact the overall performance, it is often ignored by many lock-free data structures [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. The common alternative is to rely on garbage collection systems, but in most cases these compromise the lock-freedom property of the system as a whole [16, 17, 18].

The need for SMR methods was first reported by Kung and Lehman [19] while developing a concurrent binary tree data structure supporting non-blocking search operations. Since then, multiple SMR methods have been developed with differing trade-offs in terms of capabilities, performance characteristics and hardware requirements [20, 21, 22, 23, 24, 25, 26, 27, 17, 28, 29, 18, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]. As such, nowadays it is vital to understand the requirements of the SMR methods before starting the creation of the design of the lock-free data structure, as the SMR method used will impact not only the ability

of the data structure to hold the lock-freedom property when integrated in a complete system, but will also impact the performance and memory usage of the system as a whole.

However, an ideal SMR method is impossible to achieve as recently proved by the ERA theorem [45], which states that from the three key properties for an SMR method: (i) ease of use, which determines if a method can easily be applied to any data structure; (ii) robustness, which determines the maximum amount of unreleased memory a method allows; and (iii) applicability, which determines if the method is compatible with any data structure; only two of these three properties can be achieved by any specific SMR method.

For example, by ensuring that a lock-free data structure design follows some properties, we can allow SMR methods that lack the applicability property to be compatible with the data structure, e.g., the well-known Hazard Pointers method [25] lacks applicability but is compatible with data structures that do not traverse invalidated nodes. And, by having simpler designs, we allow the usage of methods that lack the ease of use property without too much effort. SMR methods that lack the ease of use property will always be difficult to apply to a specific data structure, but the complexity of the data structure can severely exacerbate this difficulty. For example, the Optimistic Access method [30] lacks the ease of use property due to requiring rollbacks. Therefore, the more rollback points a data structure algorithm needs, the more difficult it will be to apply the method to the data structure. The use of methods that are both applicable and easy to use, such as Epoch Based Reclamation [24], does not impose constraints on data structure design. However, their lack of robustness implies that, in practice, the system as a whole may eventually block memory allocation due to exhaustion.

In short, the way a given data structure is designed will condition the type and number of SMR methods that can be chosen and applied. In particular, this allows the choice of the methods that show better performance characteristics, or the rapid testing of multiple methods in order to experimentally verify which one performs better in a particular domain.

In this work, we present a lock-free hash map design as an example that is both: (i) compatible with SMR methods that lack the applicability property; and (ii) simple enough that it allows the usage of SMR methods that lack the ease of use property without much effort. Our design is based on a previous sophisticated lock-free hash map design, named *Lock-Free Hash Tries (LFHT)* [46], that is much more complex and also incompatible with most SMR methods [47]. We show that, with a simpler redesign of the LFHT data structure, it is possible to achieve compatibility with all SMR methods and also better performance.

Our new approach, named *Sleek Lock-Free Hash Tries (SLFHT)*, leverages the characteristics of multicore cache-based architectures to simplify the implementation, make the data structure compatible with any SMR method, make the data structure more cache friendly, and solve some disadvantages of the original LFHT's design.

The remainder of the paper is organized as follows. First, we introduce some background regarding the original LFHT design. Next, we discuss the main aspects of the new SLFHT design, we describe the key algorithms required to easily reproduce our implementation by others, and we present the formal proof that our proposal is linearizable and lock-free. Then, we discuss how SMR methods can now be supported in the SLFHT design. Finally, we show experimental results, and we end by summarizing our work.

2. Original Lock-Free Hash Tries Design

The LFHT data structure has two kinds of nodes: *hash nodes* and *leaf nodes*. The leaf nodes store key/-value pairs and the hash nodes implement a hierarchy of hash levels, each node with a fixed size bucket array of 2^w entries. To map a key/value pair (k,v) into this hierarchy, a hash value h is computed for k and then chunks of w bits from h are used to index the appropriate hash node, i.e., for each hash level H_i, the i^{th} group of w bits of h are used to index the entry in the appropriate bucket array of H_i.

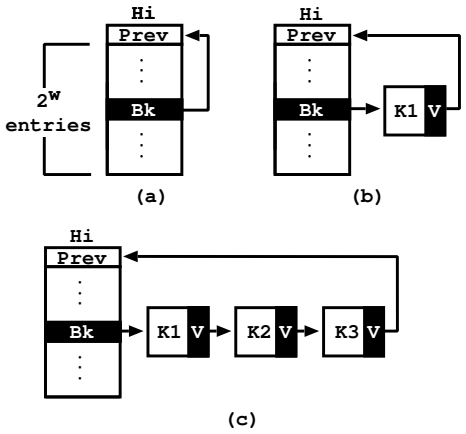


Figure 1: Insertion of nodes in a hash level

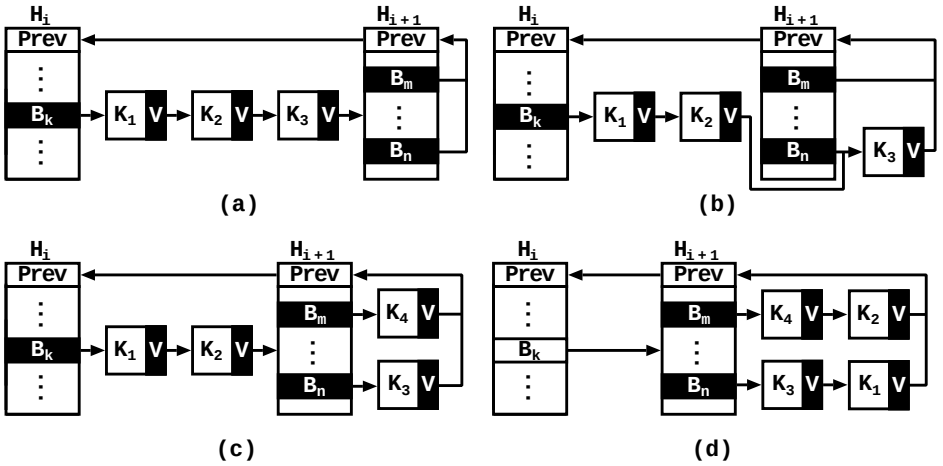


Figure 2: Expansion of nodes in a hash level

When leaf nodes fall on the same bucket entry because their hash value is sufficiently similar, we say that they *collide*. In such cases, they form a collision chain within the corresponding bucket entry, which is implemented as a linked list. Once the length of a collision chain exceeds a predefined threshold, an expansion operation is triggered, redistributing the nodes into a new hash level H_{i+1}, i.e., instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w. Figure 1 shows how the insertion of nodes is done in a hash level.

Figure 1(a) shows the initial configuration for a hash level. Each hash level is formed by a hash node H_i, which includes a bucket array of 2^w entries and a backward reference *Prev* to the previous hash level, and by the corresponding chain of nodes per bucket entry. Initially, all bucket entries are empty (B_k represents a particular bucket entry of H_i). A bucket entry stores either a reference to a hash node (initially the current hash node) or a reference to a separate chain of leaf nodes, corresponding to the hash collisions for that entry. Figure 1(b) shows the configuration after the insertion of node K_1 on B_k and Fig. 1(c) shows the configuration after the insertion of nodes K_2 and K_3. A leaf node holds both a reference to a next-on-chain node and a flag with the condition of the node, which can be valid (V) or invalid (I). The last leaf node in the chain references back the current hash node. When the number of valid nodes in a chain reaches a given threshold, the next insertion causes the corresponding bucket entry to be expanded to a new hash level. Figure 2 shows how nodes are expanded.

The expansion operation starts by inserting a new hash node H_{i+1} at the end of the chain with all its bucket entries referencing H_{i+1} and the *Prev* field referencing H_i (as shown in Fig. 2(a)). From this point on, new insertions will be done on the new level H_{i+1} and the chain of leaf nodes on H_i will be moved, one at a time, to H_{i+1}. Figure 2(b) and Fig. 2(c) show how node K_3 is first mapped in H_{i+1} (bucket B_n) and then moved from H_i (bucket B_k). It also shows a new node K_4 being inserted simultaneously by another thread. When the last node is moved, the bucket entry in H_i references H_{i+1} and becomes immutable (Fig. 2(d)). Immutable fields are represented with a white background.

Next, Fig. 3 shows an example illustrating how a node is removed from a chain. The remove operation can be divided in two steps: (i) the invalidation of the node (shown in Fig. 3(a)) and (ii) making the node unreachable (shown in Fig. 3(b)). The invalidation step starts by finding the node N (node K_2 in Fig. 3) to be removed and by changing its flag from valid (V) to invalid (I). If the flag is already invalid, it means that another thread is also removing the node and, in such case, nothing else needs to be done. Next, to make the node unreachable, the two valid nodes A and B, after and before N respectively, need to be found. The valid node A is first found, node K_3 in Fig. 3 (note that, if N is the last node in the chain, A would be a hash node). Then, by continuing to traversing the chain, a hash node H_i is found (if not yet found in the previous step). Now, if H_i is the same hash node that the traversal of the chain started from, the chain is traversed again until the last valid node B before N is found, node K_1 in Fig. 3 (or we consider B the bucket entry, if no valid node exists before N). If, while searching for B node N is not found, it means that N has already been made unreachable and the removal is complete. Otherwise, we just need to change the ref-

erence of B to A. This is shown in Fig. 3(b), where K_1 is made to refer to K_3.

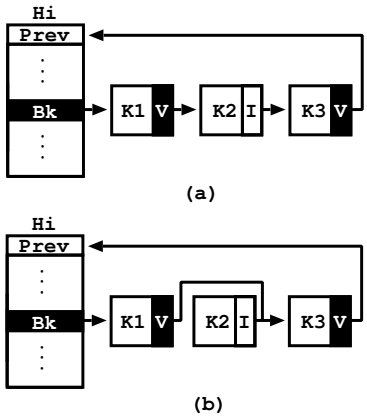


Figure 3: Removal of nodes in a hash level

On the other hand, if H_i is not the same hash node the traversal of the chain started from, this means that a concurrent expansion operation is happening simultaneously and, in such situation, the thread doing the expansion becomes responsible for making N unreachable if it sees N as invalid after moving it to the next level. This means that, during an expansion operation, nodes being removed by a thread can be momentarily reinserted in the next level by the expanding thread, which can be a problem since most SMR methods rely on the fact that a node is permanently unreachable after it is removed. This temporary reinsertion of nodes during an expansion operation required the development of an SMR method specific to the LFHT data structure, named *Hazard Hash and Level (HHL)*, which explores the characteristics of the LFHT structure to achieve efficient memory reclamation with low and well-defined memory bounds [47].

3. New Design by Example

To deal with collisions on a bucket entry, the LFHT design chains leaf nodes in a linked list up to a certain threshold, at which point a new insertion triggers an expansion that inserts a new hash node and moves the leaf nodes one by one to the new hash level. The key idea behind the new SLFHT design is to replace these collision chains with an array of leaf nodes. This change transforms what would be random memory accesses while traversing the chain list into sequential accesses, thus taking advantage of the fact that, in modern architectures, a memory access causes an entire cache line to be loaded (and often more than one cache line due to prefetching optimizations at the hardware level). So, leaf nodes become an array with a header that specifies the number of nodes in the array, followed by the nodes that collide in the corresponding bucket entry sequentially in memory. This change also saves memory, as we no longer need a reference field to the next node for every leaf node, but only a header field per group of nodes in an array. In what follows, we call these arrays of leaf nodes as *leaf arrays*.

The insertion procedure, instead of adding a node to the chain, now replaces the entire leaf array with a new one containing all the previous leaf nodes plus the new node. Figure 4 shows an example of two nodes being inserted in a hash node using leaf arrays. We start with an empty hash node H_i in Fig. 4(a). Then, in Fig. 4(b), we insert node K_1 by adding a leaf array with size 1 and node K_1. Next, in Fig. 4(c), as node K_2 collides on bucket B_k, we replace the previous leaf array with a new one with size 2 that contains both K_1 and K_2.

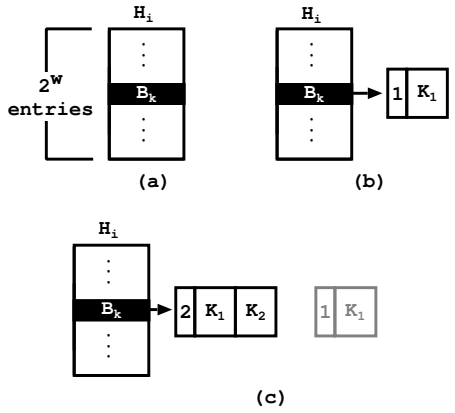


Figure 4: Insertion of nodes in a hash level using leaf arrays

Similarly, the removal procedure replaces the entire leaf array with a new one that contains all the previous nodes except the one being removed. Figure 5 shows an example of the removal of the nodes K_1 and K_2 from H_i. We show the initial state in Fig. 5(a). Then, in Fig. 5(b), we remove K_2 by replacing the leaf array containing K_1 and K_2 with a new leaf array that only contains K_1. Next, in Fig. 5(c), we remove K_1 by removing the entire leaf array as there are no nodes left to keep in B_k.

Both insertion/removal procedures are implemented using a CAS (compare-and-swap) operation, which ensures that the corresponding procedure only succeeds when no other thread replaced the leaf array in the meantime, and that, in case of failure, we simply need to retry the insertion/removal procedure. The CAS opera-

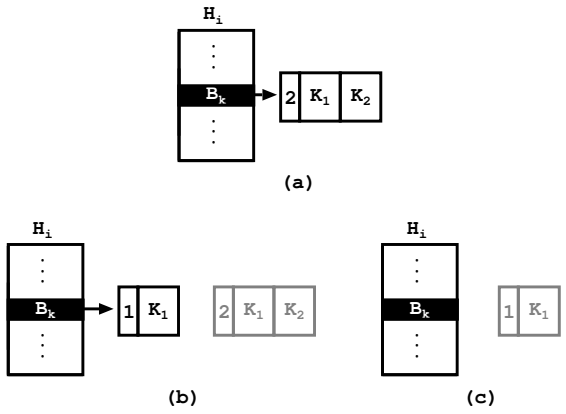


Figure 5: Removal of nodes in a hash level using leaf arrays

tion is at the heart of many lock-free data structures. It is implemented by an *atomic instruction* that compares the contents of a memory location to a given old value and, if they are the same, updates the contents of that memory location to a given new value. The atomicity guarantees that each CAS operation is performed based on up-to-date information, i.e., if the previous old value had been updated by another thread in the meantime, the current CAS would fail.

As in the case of the original LFHT design, when the number of collisions reaches a certain threshold, we trigger an expansion. The expansion is now also much simpler, as the expanding thread can create locally the new hash node with the necessary new leaf arrays and then replace with a single CAS the old leaf array with the new hash node. In the case of a CAS failure, we retry the operation from the beginning, which might not trigger an expansion anymore if either another thread also performed an expansion or performed a removal of a node from the leaf array (thus allowing us to insert a node without expanding).

Figure 6 shows an example of an expansion triggered by the insertion of a node K_4 that falls in bucket B_k (we assume an expansion threshold of 3). We start the expansion by creating a new hash node H_{i+1} (Fig. 6(a)). Then, we locally insert into H_{i+1} the nodes present in the leaf array of B_k (Fig 6(b)). Next, we replace the leaf array in B_k by the hash node H_{i+1} with all the nodes pre-inserted (Fig 6(c)). Finally, we can insert node K_4 by replacing the leaf array in bucket B_n (we assume that node K_4 now falls in bucket B_n) by a new leaf array containing both K_2 and K_4 (Fig 6(d)).

With the new design, as we no longer need to keep references to the previous hash node and to the current hash level in the header of the hash nodes, we removed the header altogether from the hash nodes. Note that the hash level can always be inferred from context as we can no longer end up in a different hash node by following a chain of leaf nodes, due to leaf arrays not having references to other nodes. So, hash nodes become just an array of bucket entries. This change not only saves memory in the hash node, but also allows for a faster traversal, as we no longer need to read the header but only the desired bucket entry.

An important disadvantage of the new SLFHT design is that, even though it ends up consuming less memory, we now cycle through a lot more memory during execution. In SLFHT, insertions result in memory to be reclaimed (when a leaf array is replaced the old one needs to be reclaimed) and removes result in memory allocations (when a leaf array contains more than one node, a new one needs to be allocated to contain the remaining nodes). We argue that, with modern memory allocators supporting thread-local caches and, as discussed in Section 6, with modern SMR methods, these additional costs can be largely mitigated.

4. Algorithms

In this section, we present in more detail the key algorithms supporting the SLFHT design. We start with the *Find()* procedure in Alg. 1 that shows how the data structure is traversed and is then used to support the *Search()*, *Insert()* and *Remove()* procedures presented in Alg. 2, 3 and 4, respectively.

Algorithm 1 $Find(node\ Hn,\ level\ l,\ hash\ h,\ key\ k)$

```
1: A ← Hn[Index(h, l)]
2: if NodeType(A) = HASH_NODE
3:     return Find(A, l+1, h, k)
4: if A ≠ Null
5:     for i ← 0 to A.size
6:         if A.node[i].hash = h ∧ A.node[i].key = k
7:             return ⟨A.node[i].value, Hn, A, l, i⟩
8: return ⟨Null, Hn, A, l, Null⟩
```

The $Find()$ procedure takes four arguments: (i) the base hash node (where to start the traversal); (ii) the level the base hash node is at (note that the level needs to be passed as an argument as it is no longer present in the header of the hash node); (iii) the hash or insertion point to be found; and (iv) the key or insertion point to be found.

The $Find()$ procedure is used for two purposes: (i) to find the leaf node corresponding to the given hash

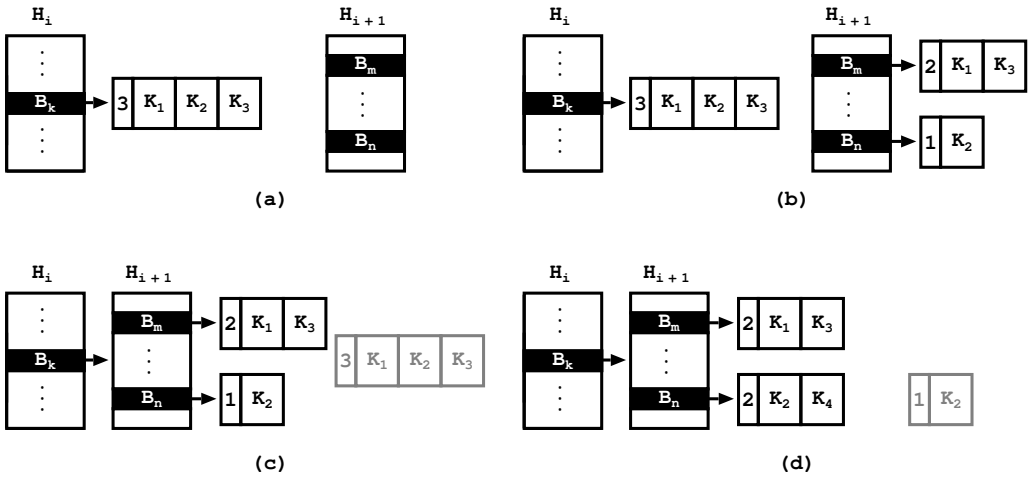


Figure 6: Expansion of nodes in a hash level using leaf arrays

and key arguments; or (ii) to find the hash node where the given hash and key arguments should be inserted. It then returns the following values: (i) the value stored in the leaf node corresponding to the given hash and key arguments, or *Null* if no leaf node is found; (ii) the hash node corresponding to the leaf node found or to the insertion point; (iii) the leaf array corresponding to the leaf node found or to the insertion point, or *Null* if no leaf array is found; (iv) the level of the hash node being returned; and (v) the index in the leaf array corresponding to the leaf node found, or *Null* if no leaf node is found. The *Index*() auxiliary function extracts the relevant bits from the hash argument based on the level, and the *NodeType*() auxiliary function extracts the type of the node from the least significant bit of the given reference.

The *Search*() procedure in Alg. 2 starts by computing the hash for the given key argument and then uses it to find the corresponding node, returning the value in such node, in case of success, or *Null* in case no such node exists.

Algorithm 2 *Search*(*node Hn*, *key k*)

```
1: h ← Hash(k)
2: ⟨r, Hn, A, l, i⟩ ← Find(Hn, 0, h, k)
3: return r
```

The *Insert*() procedure in Alg. 3 executes in a loop until it is either able to insert a new node for the given arguments (lines 12 – 14) or to find a node already present in the data structure corresponding to the given arguments (lines 5 – 6). If, at any point, the leaf array it is trying to insert on is full (line 7), it first performs an expansion (lines 8 – 10) and then retries the insertion. Note that if we are at the last level, the full hash has been consumed and as such all nodes present in the leaf array are full hash collisions. Also note that the *CreateExpandHash*() auxiliary function creates a new hash node *N* with the nodes present in the given leaf array *A* already inserted in the respective buckets (line 8), and the *CreateAddNode*() auxiliary function creates a new leaf array *N* with the new node in addition to the contents of the previous leaf array *A* (line 12).

Algorithm 3 *Insert*(*node Hn*, *key k*, *value v*)

```
1: h ← Hash(k)
2: l ← 0
3: loop
4:    ⟨r, Hn, A, l, i⟩ ← Find(Hn, l, h, k)
5:    if r ≠ Null
6:       return ⟨False, r⟩
7:    if A ≠ Null ∧ A.size = THRESHOLD ∧
      i < LAST_HASH_LEVEL
8:       N ← CreateExpandHash(l + 1, A)
9:       if CAS(Hn[Index(h,l)], A, N)
10:         Hn ← N
11:   else
12:      N ← CreateAddNode(A, h, k, v)
13:      if CAS(Hn[Index(h, l)], A, N)
14:         return ⟨True, v⟩
```

The *Remove*() procedure in Alg. 4 executes in a loop until it is either able to remove the node corresponding to the given key (lines 7 – 9) or not able to find such a node (lines 5 – 6). The *CreateRemoveNode*() auxiliary function creates a new leaf array *N* with the contents of

the previous leaf array A minus the node at the specified index (line 7). [1]

Algorithm 4 *Remove*(*node Hn*, *key k*)

```
1: h ← Hash(k)
2: l ← 0
3: loop
4:    ⟨r, Hn, A, l, i⟩ ← Find(Hn, l, h, k)
5:    if r = Null
6:       return ⟨False, Null⟩
7:    N ← CreateRemoveNode(A, i)
8:    if CAS(Hn[Index(h, l)], A, N)
9:       return ⟨True, r⟩
```

5. Correctness

In this section, we discuss the correctness of the SLFHT design. The formal proof consists of two parts: first prove that the design is linearizable and then prove that progress occurs in a lock-free fashion for all operations.

5.1. Linearizability

Linearizability is an important correctness condition for the implementation of concurrent data structures [48]. Linearizability ensures the correctness of concurrent data structures by proving that semantically consistent (non-interfering) operations may execute in parallel. An operation is linearizable if it appears to take effect instantaneously at some moment of time t_{LP} between its invocation and response. Literature often refers to t_{LP} as a linearization point and, for lock-free implementations, a linearization point is typically a single instant where its effects become visible to all the remaining operations. Linearizability guarantees that if all operations individually preserve an invariant (or a set of invariants), the system as a whole also will.

Our proof has three parts. First, we enumerate the linearization points of the SLFHT design. Second, we describe the set of invariants that define a correct state for the SLFHT design. Third, we prove that every linearization point preserves the set of invariants. We thus begin by enumerating the linearization points within the algorithms:

[1] Note that both *CreateAddNode*() and *CreateRemoveNode*() auxiliary functions consider that an empty leaf array is represented by a *Null* reference.

LP_1 The *Insert()* procedure (Alg. 3) is linearizable at successful CAS in line 9.

LP_2 The *Insert()* procedure (Alg. 3) is linearizable at successful CAS in line 13.

LP_3 The *Remove()* procedure (Alg. 4) is linearizable at successful CAS in line 8.

Next, we describe the set of invariants that must be *preserved* on every state of the SLFHT design:

Inv_1 The initial reference of every bucket entry B belonging to the root hash level is *Null*.

Inv_2 The reference stored in a bucket entry B belonging to a hash level H_i must be one of the following: (i) *Null*; (ii) a reference to a leaf array; or (iii) a reference to a deeper hash level H_{i+1}.

Inv_3 For a leaf array A in a bucket entry B belonging to a hash level H_i ($i < LAST\_HASH\_LEVEL$), the number of leaf nodes in A is always lower than or equal to a predefined *THRESHOLD* value ($THRESHOLD \geq 1$).

Next, we prove that every linearization point *preserves* the set of invariants.

Lemma 1. *In the initial state of the data structure the set of invariants hold.*

Proof. In the initial state, the root hash level is empty, i.e., all bucket entries refer to *Null*, and thus all invariants hold. □

Lemma 2. *The linearization point LP_1 preserves the set of invariants.*

Proof. A successful CAS operation in the linearization point LP_1 (Alg. 3, line 9) updates a bucket entry B to a hash level N. For the CAS operation to be triggered, then the thread executing the *Insert*() procedure has found a leaf array A in a bucket entry B belonging to a hash level H_i with $i < LAST\_HASH\_LEVEL$, and A has a number of nodes equal to *THRESHOLD* (line 7). The thread then created a new hash node N with all the leaf nodes present in A pre-inserted in N (line 8). As such, Inv_1 is not affected, the resultant update of B is a reference to a hash level (Inv_2 holds) and each bucket entry in the newly created hash node N is referring to either *Null* or to a leaf array with a number of leaf nodes lower or equal to *THRESHOLD* (Inv_3 holds). □

Lemma 3. *The linearization point LP_2 preserves the set of invariants.*

Proof. A successful CAS operation in the linearization point LP_2 (Alg. 3, line 13) updates a bucket entry B to a leaf array N. For the CAS operation to be triggered, then the thread executing the $Insert()$ procedure has found: (i) an empty bucket entry B; (ii) a leaf array A with a number of leaf nodes lower than *THRESHOLD*; or (iii) a leaf array with a number of leaf nodes greater or equal to *THRESHOLD* with B belonging to a hash level H_i with $i = LAST\_HASH\_LEVEL$ (check the condition in line 7), and has created a new leaf array N that includes the key being inserted in the data structure. As such, Inv_1 is not affected, Inv_2 holds because N is a leaf array and Inv_3 also holds since N must have a number of leaf nodes which is lower or equal to *THRESHOLD* unless N is at the last level. □

Lemma 4. *The linearization point LP_3 preserves the set of invariants.*

Proof. A successful CAS operation in the linearization point LP_3 (Alg. 4, line 8) updates a bucket entry B to *Null* or to a leaf array N. For the CAS operation to be triggered, then the thread executing the $Remove()$ procedure has found a leaf array A in a bucket entry B and has updated B to a newly created leaf array N which does not include the leaf node for the key being removed or to *Null* (case A only contains the leaf node for the key being removed). As such, Inv_1 is not affected, Inv_2 holds because B is either updated to a leaf array or to *Null*, and Inv_3 also holds because N must have a number of leaf node which is lower than *THRESHOLD* unless N belongs to a hash node in the last level. □

Corollary 1. The invariants hold on every configuration of the SLFHT design due to Lemmas 1 to 4.

Theorem 5. *The SLFHT design is linearizable.*

5.2. Lock-Freedom

We next prove the lock-freedom property of the SLFHT design. The proof of the progress of threads in non-linearization points is trivially shown, as instructions allow threads to execute non-blocking operations freely within the data structure. For the progress in the linearization points we have:

Lemma 6. *When a thread executes the operations defined by the linearization points LP_1, LP_2 and LP_3 then the configuration of the data structure has change and at least one thread has made progress.*

Proof. All linearization points correspond to CAS operations on a given memory location M trying to update an initial reference R_i to a secondary reference R_s. As shown in the previous subsection, R_i and R_s must always be a reference to *Null*, a leaf array or a hash level. Assuming that t_i is the instant of time where a thread T first reads a reference R_i from M and that t_f is the instant of time where T executes the CAS operation trying to update R_i to R_s, then a successful CAS execution leads to progress in the state of the data structure as M was updated to R_s. Otherwise, if the CAS operation fails, that means that between instants t_i and t_f, the reference on M was changed, which means that at least another thread has changed M between the instants of time t_i and t_f, thus leading to progress in the state of the data structure. □

Theorem 7. *The SLFHT design is lock-free.*

6. Safe Memory Reclamation

An important disadvantage of the SLFHT data structure is its need to cycle through much more memory during execution. Having leaf nodes represented sequentially in memory as leaf arrays results in extra memory allocations and in extra memory reclamations when inserting/removing nodes. In particular, when replacing a leaf array with a new one, we need to reclaim the memory of the old one. To reclaim the memory of a leaf array, extra care needs to be taken to ensure that no thread possesses an old reference to it, at the consequence of reading invalid memory, corrupting the process memory or triggering an access violation from the operating system.

SMR methods are responsible for tracking references to retired memory (memory that is no longer reachable from the data structure but might still be reachable from thread local references) and making such memory available for reuse when it is safe to do so. Some of the most important properties for an SMR method are: (i) *ease of use*, the ability to be easy to apply to any data structure; (ii) *robustness*, the ability to ensure a bound on the amount of memory that has been retired but not made available for reuse; and (iii) *applicability*, the ability to be used on any data structure algorithm.

In recent work, Sheffi and Petrank introduced the ERA theorem [45] where it is proved that any specific SMR method can only have two of these three properties. Since there is no ideal SMR method and garbage collection is not an option to ensure lock-freedom, there is an increasing pressure to design data structures compatible with most SMR methods, thus allowing the usage of methods that lack the *ease of use* and/or *applicability* properties.

Most often, an SMR method lacks the *ease of use* property if: (i) it requires rollbacks; or (ii) it requires changes to the data structure fields. Rollbacks impose an extra complexity layer since the user has to consider what are the safe places to rollback to, and fields changes require the user to reconsider how the data structure behaves when such changes happen. In order to mitigate these difficulties, one can argue that the best course of action is to make the algorithms of the data structure as simple as possible, to facilitate the reasoning about safe points and/or fields changes.

The main factor that makes a SMR method that lacks *applicability* incompatible with a data structure is the case where the target data structure needs to traverse invalid nodes. A well-known example is Michael's proposal [25] for the Harris lists [23] that avoids the traversal of invalid nodes in order to make it compatible with the Hazard Pointers method (which lacks applicability). This is an important property to achieve when developing a data structure as it will allow compatibility with a much wider range of SMR methods.

A data structure that requires both the ease of use and applicability properties can only be used with a non-robust method. By designing the data structure to forgo at least one of these two requirements, we ensure compatibility with robust SMR methods.

The sleekness of the SLFHT design stems from its new leaf array representation, which prevents LFHT's original traversal of invalid nodes, makes SLFHT compatible with both SMR methods that lack the ease of use property and/or that lack the applicability property. As a way of showing this, we implemented two methods for the SLFHT design that are representative of SMR methods that lack the easy of use and applicability properties.

As a first option, we chose the *Hazard Pointers (HP)* method, proposed by Michael [25], as it is one of the most commonly used, and serves as an example of a method that lacks applicability. Moreover, the HP method also tends to achieve good performance in data structures with low depth and has tight memory bounds in our use case. As a second option, we chose the *Optimistic Access (OA)* method that was developed by Cohen and Petrank [30], and then further improved by Moreno and Rocha [49]. The OA method is an example of a method that lacks the ease of use property, and is also one of the most efficient memory reclamation methods while being robust and applicable. Note that both HP and OA methods have the *robustness* property, i.e., they ensure a bound on the amount of memory that has been retired but not made available for reuse. This is a key property since otherwise we cannot guarantee that, when running our data structure, it will not exhaust the available memory and take away the lock-freedom property of the system as a whole.

6.1. Hazard Pointers

The HP method works by setting a hazard pointer to the node that we want to access in order to inform the other threads that such node should not be reclaimed. Each thread has a set of hazard pointers that is visible by all threads but only modifiable by the thread that owns it. When performing reclamation, a thread verifies if there is any hazard pointer protecting the node it wants to reclaim and, in case there is not, it can safely reclaim the node, otherwise the reclamation of the node is postponed until the next reclamation.

In SLFHT, we only need one hazard pointer per thread as only leaf arrays need to be protected. This allows us to only apply the expensive hazard pointer protection to leaf arrays, which on average happens once per operation. This allows for the HP method to be extremely efficient when applied to the SLFHT data structure.

Thanks to its simple design, integrating the HP method into the SLFHT data structure is straightforward. In the following, we show how the $Find()$, $Insert()$, and $Remove()$ procedures were adapted to support hazard pointers.

In the $Find()$ procedure, shown in Alg. 5, it is sufficient to protect the current leaf array A by assigning a hazard pointer to it (line 5), and then verify that the leaf array remains reachable by reading its initial value (lines $6-7$). This check is essential because the array may have been removed and reclaimed in the meantime, in which case the procedure restarts from the current hash node (line 8).

Algorithm 5 $Find(node\ Hn,\ level\ l,\ hash\ h,\ key\ k)$

```
1:  A ← Hn[Index(h, l)]
2:  if NodeType(A) = HASH_NODE
3:      return Find(A, l + 1, h, k)
4:  if A ≠ Null
5:      HP ← A
6:      A ← Hn[Index(h, l)]
7:      if A ≠ HP
8:          return Find(Hn, l, h, k)
9:      for i ← 0 to A.size
10:         if A.node[i].hash = h ∧ A.node[i].key = k
11:             return ⟨A.node[i].value, Hn, A, l, i⟩
12: return ⟨Null, Hn, A, l, Null⟩
```

Regarding the $Insert()$ and $Remove()$ procedures, shown in Algs. 6 and 7, respectively, it is only neces-

sary to add the appropriate *Retire*() calls (lines 10 and 15 in Alg. 6, and line 9 in Alg. 7) to move the node to the corresponding thread's retire list, where it awaits safe reclamation.

Algorithm 6 *Insert*(*node Hn*, *key k*, *value v*)

```
1:  h ← Hash(k)
2:  l ← 0
3:  loop
4:    ⟨r, Hn, A, l, i⟩ ← Find(Hn, l, h, k)
5:    if r ≠ Null
6:      return ⟨False, r⟩
7:    if A ≠ Null ∧ A.size = THRESHOLD ∧
      i < LAST_HASH_LEVEL
8:      N ← CreateExpandHash(l + 1, A)
9:      if CAS(Hn[Index(h,l)], A, N)
10:       Retire(A)
11:       Hn ← N
12:   else
13:     N ← CreateAddNode(A, h, k, v)
14:     if CAS(Hn[Index(h, l)], A, N)
15:       Retire(A)
16:       return ⟨True, v⟩
```

Algorithm 7 *Remove*(*node Hn*, *key k*)

```
1:  h ← Hash(k)
2:  l ← 0
3:  loop
4:    ⟨r, Hn, A, l, i⟩ ← Find(Hn, l, h, k)
5:    if r = Null
6:      return ⟨False, Null⟩
7:    N ← CreateRemoveNode(A, i)
8:    if CAS(Hn[Index(h, l)], A, N)
9:      Retire(A)
10:     return ⟨True, r⟩
```

6.2. Optimistic Access

The OA method works by optimistically reading memory locations and only then verifying if such reads were valid by checking for a warning from a reclaiming thread. In the case of receiving a warning, the method forces the current operation to restart from a safe location. As hash nodes are never removed (both in the original LFHT and the new SLFHT design), we can consider any hash node as a safe location to restart when applying the OA method. We can also forego the verification of the warning when accessing hash nodes since the node type is embedded in the reference to the node, meaning that we know the type of a node before accessing it, and thus we do not need to read the node type from the node itself, which would require a validation of such read.

On the other hand, when accessing leaf arrays, we cannot read the whole leaf array and only then verify the warning. This is because we do not know the size of the leaf array before accessing it. Consider a scenario where we start accessing the leaf array by reading its size field, but such leaf array was already reclaimed and the memory reused. In such a case, the size field could have any value, and as such, could cause us to read beyond the end of the leaf array. The general solution is to treat accesses to leaf arrays as two separate accesses, the first access reads the size and verifies if it is valid, the second access then reads the array of nodes (that also needs to be verified after). Note that even if the leaf array was reclaimed after the first access succeeds its verification, we can be sure that a leaf array with that size at that memory address has existed in the past and as such the memory must be accessible, even if the contents are invalid.

In order to remove or replace a leaf array, we just need to protect it with a hazard pointer and then verify the warning in order to make sure the hazard pointer was properly set. Note that even taking into account that the OA method lacks the *ease of use* property, the simplicity of the data structure allows the OA method to be applied without much effort.

Algorithm 8 *Find*(*node Hn*, *level l*, *hash h*, *key k*)

```
1:  A ← Hn[Index(h, l)]
2:  if NodeType(A) = HASHNODE
3:    return Find(A, l + 1, h, k)
4:  if A ≠ Null
5:    Size ← A.size
6:    if Warning()
7:      return Find(Hn, l, h, k)
8:    for i ← 0 to Size
9:      if A.array[i].hash = h ∧ A.array[i].key = k
10:       r ← A.array[i].value
11:       if Warning()
12:         return Find(Hn, l, h, k)
13:       else
14:         return ⟨r, Hn, A, l, i⟩
15: if Warning()
16:   return Find(Hn, l, h, k)
17: return ⟨Null, Hn, A, l, Null⟩
```

In the following, we show how the *Find*(), *Insert*(), and *Remove*() procedures were adapted to support the

OA method. Algorithm 8 shows the new *Find*() procedure extended with the OA method. The main modifications are the addition of new *Warning*() calls. A first call after reading the size of the leaf array (lines $5-7$), a second one after finding the node at hand (lines $10-12$), and a third one if reaching the end of the leaf array without finding the node (lines $15-16$). In all cases, if a warning is detected, the procedure restarts from the current hash node.

Regarding the *Insert*() and *Remove*() procedures, shown in Algs. 9 and 10, respectively, they include: (i) the hazard pointer protection followed by a warning verification (lines $7-8$ in Alg. 9, and lines $7-8$ in Alg. 10); and (ii) the appropriate *Retire*() calls (lines 12 and 17 in Alg. 9, and line 11 in Alg. 10) to move the node to the corresponding thread's retire list, where it awaits safe reclamation.

Algorithm 9 *Insert*(*node Hn*, *key k*, *value v*)

```
1:  h ← Hash(k)
2:  l ← 0
3:  loop
4:    ⟨r, Hn, A, l, i⟩ ← Find(Hn, l, h, k)
5:    if r ≠ Null
6:      return ⟨False, r⟩
7:    HP ← A
8:    if ¬Warning()
9:      if A ≠ Null ∧ A.size = THRESHOLD ∧ i <
        LAST_HASH_LEVEL
10:       N ← CreateExpandHash(l + 1, A)
11:       if CAS(Hn[Index(h,l)], A, N)
12:         Retire(A)
13:         Hn ← N
14:     else
15:       N ← CreateAddNode(A, h, k, v)
16:       if CAS(Hn[Index(h, l)], A, N)
17:         Retire(A)
18:         return ⟨True, v⟩
```

Algorithm 10 *Remove*(*node Hn*, *key k*)

```
1:  h ← Hash(k)
2:  l ← 0
3:  loop
4:    ⟨r, Hn, A, l, i⟩ ← Find(Hn, l, h, k)
5:    if r = Null
6:      return ⟨False, Null⟩
7:    HP ← A
8:    if ¬Warning()
9:      N ← CreateRemoveNode(A, i)
10:     if CAS(Hn[Index(h, l)], A, N)
11:       Retire(A)
12:       return ⟨True, r⟩
```

7. Performance Analysis

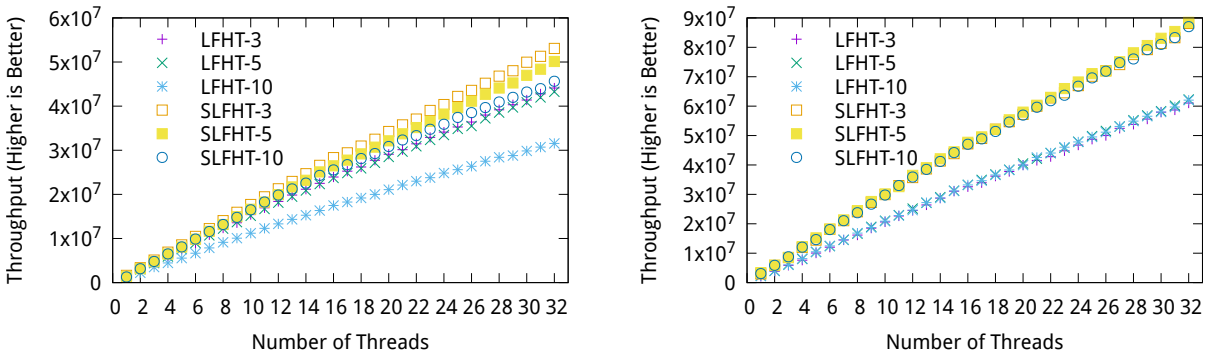
Our experimental environment was a machine with 2 x AMD Opteron™ Processor 6274 with 16 cores each, 64 B of cache line size, 16 KiB of L1 cache per core, 2048 KiB of L2 cache per two cores and 14 MiB of usable shared L3 cache per CPU, with a total of 32 GiB of DDR3 memory. The machine was running Ubuntu 22.04 with kernel 5.15.0-91 and all designs were compiled with GCC version 13.2.1 (with -O3).

For the benchmarks, we created multiple scenarios with varying ratios of search, insert and remove operations and with different configurations. We ran configurations with 2^4 and 2^8 bucket entries per hash node combined with an expansion threshold of 3, 5 and 10 nodes. All scenarios execute a total of 10^7 operations with uniformly distributed randomized keys. The use of randomized keys allows us to use the identity function ($f(x) = x$) as the hash function, reducing one variable that can affect the results (the hash function choice) while having the same expected behaviour as the usage of any key distribution with a proper hash function. There is an initial preparation stage where all keys aimed to be searched or removed during execution are pre-inserted in the data structure. To support concurrent randomicity on each thread, our benchmark tool uses *glibc PRNG*, such that, for insertions it just inserts random keys by giving each thread a different seed, and for searches and removes, it reuses the seeds used for insertion, ensuring that each key is searched or removed only once. Each scenario was ran 5 times and we took the mean of those runs. The results that follows are shown in throughput (operations per second), which is obtained by dividing the number of operations performed by the time taken to perform them.

7.1. Cache Optimization

We begin by analyzing the performance impact of the SLFHT design when compared to LFHT, both designs running without memory reclamation support and with memory allocation managed by jemalloc version 5.2.1 memory allocator [50]. The aim of this analysis is to study the design differences mainly due to the cache optimized leaf arrays introduced in SLFHT.

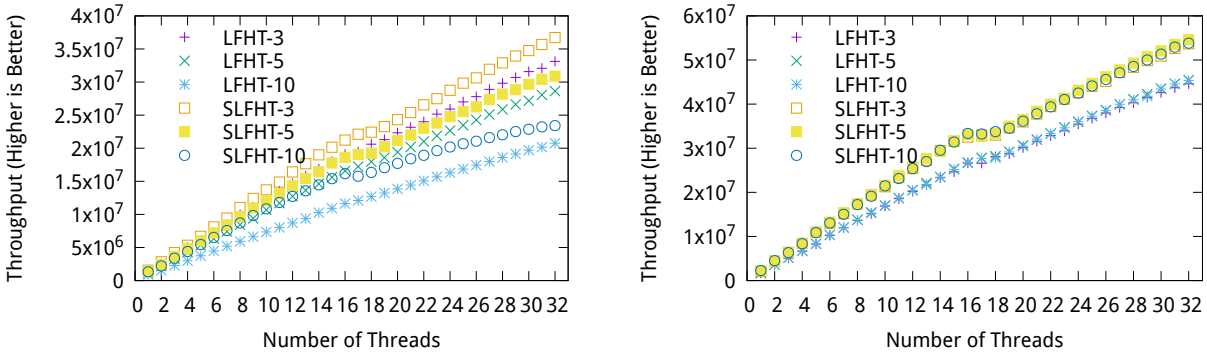
Figure 7 presents a scenario where threads execute only search operations, Fig. 8 presents a scenario where



(a) 2^4 bucket entries

(b) 2^8 bucket entries

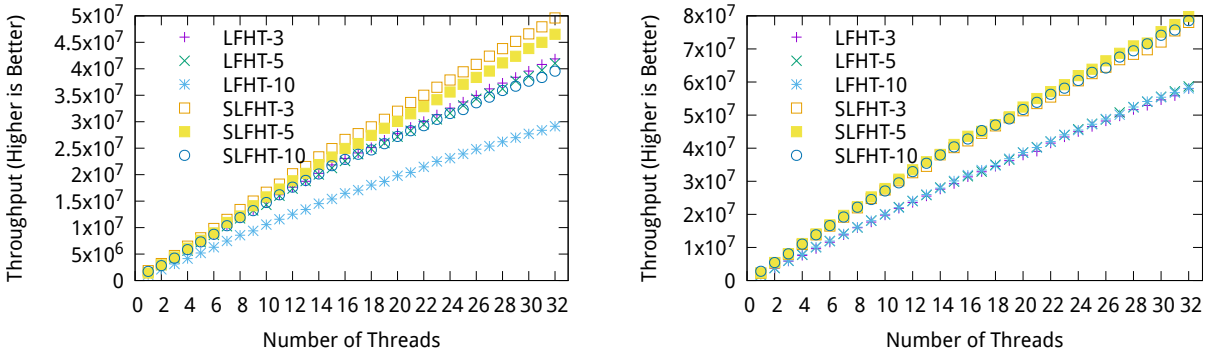
Figure 7: Throughput for the *Search Only* scenario



(a) 2^4 bucket entries

(b) 2^8 bucket entries

Figure 8: Throughput for the 50% *Inserts* and 50% *Removes* scenario

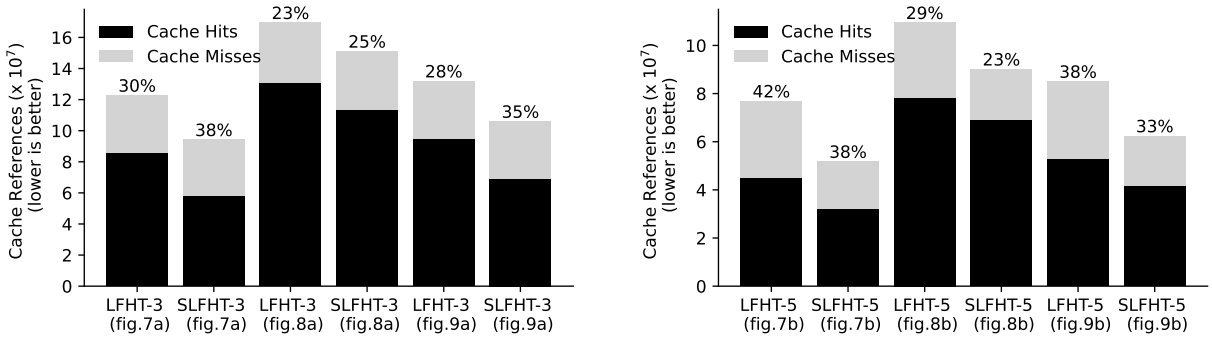


(a) 2^4 bucket entries

(b) 2^8 bucket entries

Figure 9: Throughput for the 90% *Searches*, 5% *Inserts* and 5% *Removes* scenario

threads execute 50% of search and 50% of remove operations, and Fig. 9 presents a scenario where threads execute 90% of search, 5% of insert and 5% of remove operations. The left and right sub-figures show the results obtained for configurations with 2^4 and 2^8 bucket entries per hash node, respectively. Each plot specifies the design and threshold in use, e.g., *LFHT-3* means the LFHT design with an expansion threshold of 3.



(a) 2^4 bucket entries

(b) 2^8 bucket entries

Figure 10: Number of cache references (hits and misses) with 32 threads

In all scenarios, one can observe that, for the same expansion threshold, the SLFHT design has better throughput than LFHT. The improvement is specially noticeable for larger bucket entry sizes and in benchmarks with more search operations, the best case scenario being shown in Fig. 7b where one can observe an improvement of about 42% with 32 threads with an expansion threshold of 5. Figure 8a shows that SLFHT loses some scalability with an expansion threshold of 10. We found that this was mainly caused due to the extra stress placed on the memory allocator, which was exacerbated by the fact that we were not reclaiming memory and thus, not reusing it. Even so, SLFHT still manages to outperform LFHT in all scenarios.

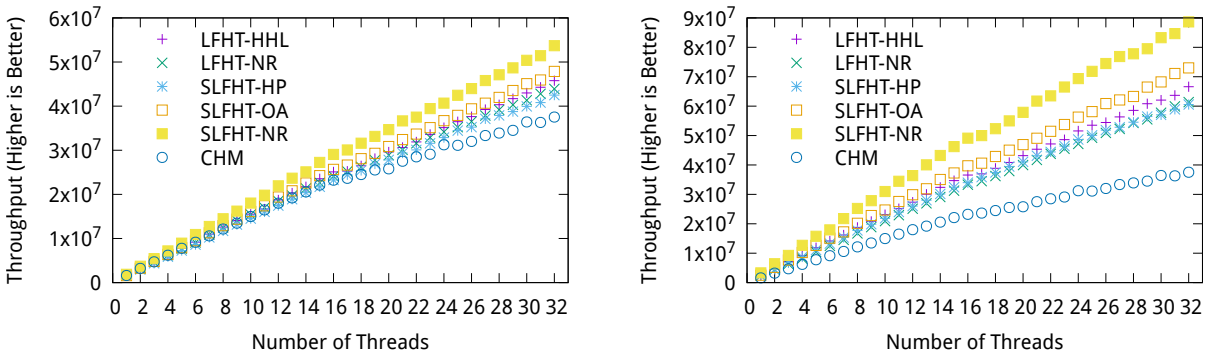
To better understand the results obtained, we used the *Linux Perf* to profile the cache references occurring during the execution of some scenarios. Figure 10 shows the number of cache references occurred on the scenarios in Fig. 7, Fig. 8 and Fig. 9, for the execution with 32 threads and for the best configuration of bucket entries and thresholds. The value shown on top of each column refers to the percentage of cache-misses occurred. One can observe that when comparing similar configurations, the SLFHT has always better performance than LFHT. The percentage of cache-misses is similar, but as the overall number of cache references is always lower on SLFHT, the number of cache-misses is also significantly lower than LFHT. This explains the better throughput results shown in the previous figures.

7.2. Memory Reclamation

We next analyze the performance impact with memory reclamation support, and evaluate SLFHT against LFHT and the Concurrent Hash Map design (CHM) of Intel-TBB library version 2021.5.0 [51]. The scenarios in Fig. 11, Fig. 12 and Fig. 13 are as before, but the plots are different as they specify the design and the safe memory reclamation method in use, e.g., *LFHT-HHL* means the LFHT design with the HHL memory reclamation method. The HHL (Hazard Hash and Level) method is the original memory reclamation method implemented for LFHT [47], the HP (Hazard Pointers) and OA (Optimistic Access) are the two SMR methods discussed in Section 6 for SLFHT, and NR are the versions without memory reclamation support. Since the SLFHT-OA version requires the usage of a compatible memory allocator in order to be able to release memory to the memory allocator/operating system [49], we used the LRMalloc memory allocator [52] for all versions.

Figure 11 illustrates the impact of the SMR method even when no memory reclamation is performed. It also shows that, in the search only scenario, SLFHT-OA achieves the best results for memory reclamation, outperforming the original LFHT-HHL version by a small margin. Curiously, in this scenario, LFHT-HHL outperforms the LFHT-NR (no reclamation) version, this is likely due to the fact that the HHL method adds a flag to the bucket entries, which allow the traversal to skip the hash node header and read the bucket entries directly. Figure 12 shows SLFHT-OA and SLFHT-HP outperforming or closely matching the SLFHT-NR (no reclamation) version. This is likely due to the effective use of the allocator thread caches, as the number of allocations and frees are closely matched with memory reclamation. This leads to fewer system calls, as the allocator needs to perform fewer memory requests to the operating system. Figure 13 shows what is probably the closest to a real world scenario, with 90% searches, 5% inserts and 5% removes, and we can still see a clear performance advantage for the SLFHT-OA version compared to all other methods that reclaim memory.

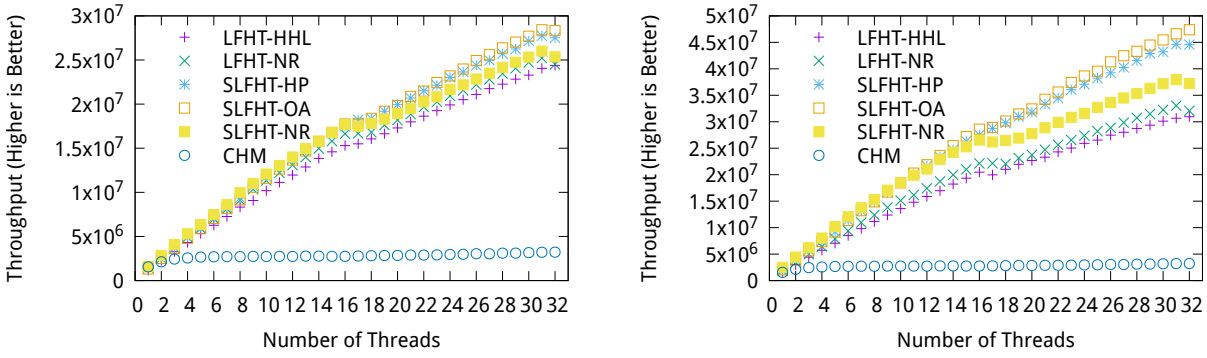
In summary, one can observe that for the SMR ver-



(a) 2^4 bucket entries and threshold 3

(b) 2^8 bucket entries and threshold 5

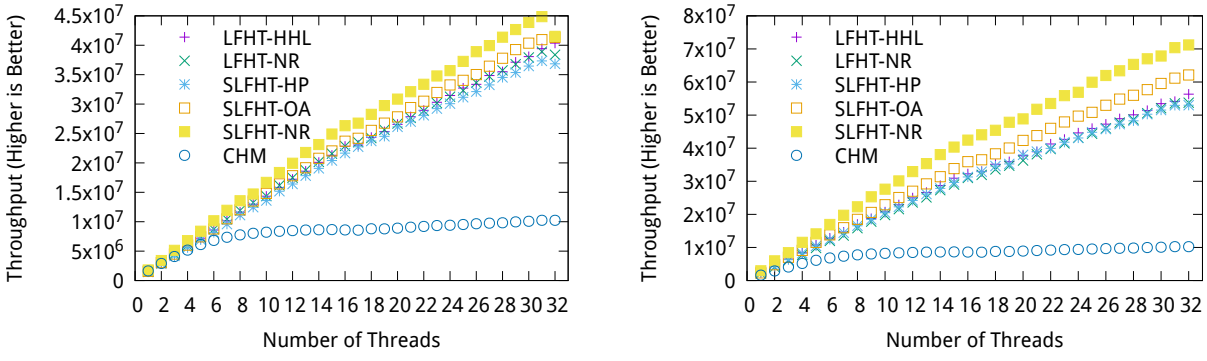
Figure 11: Throughput for the *Search Only* scenario



(a) 2^4 bucket entries and threshold 3

(b) 2^8 bucket entries and threshold 5

Figure 12: Throughput for the 50% *Inserts* and 50% *Removes* scenario



(a) 2^4 bucket entries and threshold 3

(b) 2^8 bucket entries and threshold 5

Figure 13: Throughput for the 90% *Searches*, 5% *Inserts* and 5% *Removes* scenario

sions, there is a clear gap in performance between the SLFHT and LFHT designs, and that SLFHT-OA is able to maintain an advantage in almost all scenarios. In the overall picture, when comparing against CHM, SLFHT outperforms CHM by an enormous margin. Intel's CHM here serves as a baseline for comparison as it is one of the most widely used concurrent hash map implementations.

As future work, we plan to perform additional benchmarking on more recent hardware with larger cache capacities, and to experiment with more real-world access patterns that may further highlight the benefits of our cache optimized design.

8. Final Remarks

Safe Memory Reclamation (SMR) is a technique used in concurrent programming to ensure that memory can be safely reclaimed without causing data corruption, dangling pointers, or access to freed memory. It is vital to understand the importance of properties of the SMR methods before starting the creation of the design of the lock-free data structure, as the SMR used will impact not only the ability of the data structure to hold the lock-freedom property when integrated in a complete system, but will also impact the performance and memory usage of the system as a whole.

We have redesigned the LFHT data structure in order to simplify its expansion mechanism, making it compatible with most SMR methods, and improving its performance at the same time by making it more cache friendly. Experimental results show that the new SLFHT design achieves significant performance gains, when compared against the old LFHT design and the Concurrent Hash Map design supported by Intel.

The new SLFHT design shows the importance of being aware of SMR while designing lock-free data structures. The new SLFHT design is simpler, more compatible and easy to integrate with SMR methods, making it more attractive to be used in real world applications, more reliable as it leaves less chance for mistakes during implementation, and allows for novel features to be more easily added to it.

References

[1] H. Sundell, P. Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, in: International Symposium on Parallel and Distributed Processing, IEEE Computer Society, 2003, pp. 11 pp.–.

[2] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A provably correct scalable concurrent skip list, in: International Conference on Principles of Distributed Systems, Technical Report, Bordeaux, France, 2006.

[3] O. Shalev, N. Shavit, Split-ordered lists: Lock-free extensible hash tables, Journal of the ACM 53 (3) (2006) 379–405.

[4] D. Dechev, P. Pirkelbauer, B. Stroustrup, Lock-free dynamically resizable arrays, in: Principles of Distributed Systems, Springer, 2006, pp. 142–156.

[5] M. Spiegel, P. F. Reynolds Jr., Lock-free multiway search trees, in: International Conference on Parallel Processing, 2010, pp. 604–613.

[6] F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel, Non-blocking binary search trees, in: Symposium on Principles of Distributed Computing, ACM, 2010, p. 131–140.

[7] T. Brown, J. Helga, Non-blocking k-ary search trees, in: International Conference on Principles of Distributed Systems, Springer, 2011, p. 207–221.

[8] A. Prokopec, N. G. Bronson, P. Bagwell, M. Odersky, Concurrent tries with efficient non-blocking snapshots, in: Symposium on Principles and Practice of Parallel Programming, ACM, 2012, pp. 151–160.

[9] A. Braginsky, E. Petrank, A lock-free b+tree, in: Symposium on Parallelism in Algorithms and Architectures, ACM, 2012, p. 58–67.

[10] C. M. Kirsch, M. Lippautz, H. Payer, Fast and scalable, lock-free k-fifo queues, in: Parallel Computing Technologies, Springer, 2013, pp. 208–223.

[11] A. Natarajan, N. Mittal, Fast concurrent lock-free binary search trees, in: Symposium on Principles and Practice of Parallel Programming, ACM, 2014, p. 317–328.

[12] N. Nguyen, P. Tsigas, Lock-free cuckoo hashing, in: International Conference on Distributed Computing Systems, IEEE Computer Society, 2014, p. 627–636.

[13] D. Alistarh, J. Kopinsky, J. Li, N. Shavit, The spraylist: a scalable relaxed priority queue, in: Symposium on Principles and Practice of Parallel Programming, ACM, 2015, p. 11–20.

[14] A. Prokopec, Cache-tries: Concurrent lock-free hash tries with constant-time operations, in: Symposium on Principles and Practice of Parallel Programming, ACM, 2018, pp. 137–151.

[15] K. Winblad, K. Sagonas, B. Jonsson, Lock-free contention adapting search trees, Transactions on Parallel Computing 8 (2) (2021).

[16] E. Petrank, Can parallel data structures rely on automatic memory managers?, in: Workshop on Memory Systems Performance and Correctness, ACM, 2012, p. 1.

[17] A. Braginsky, A. Kogan, E. Petrank, Drop the anchor: Lightweight memory management for non-blocking data structures, in: Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2013, p. 33–42.

[18] T. A. Brown, Reclaiming memory for lock-free data structures: There has to be a better way, in: ACM Symposium on Principles of Distributed Computing, ACM, 2015, p. 261–270.

[19] H. T. Kung, P. L. Lehman, Concurrent manipulation of binary search trees, Transactions on Database Systems 5 (3) (1980) 354–382.

[20] J. Valois, Lock-free linked lists using compare-and-swap, in: Symposium on Principles of Distributed Computing, ACM, pp. 214–222.

[21] M. Michael, M. Scott, Correction of a memory management method for lock-free data structures, Tech. rep., Rochester University, NY, Department of Computer Science (1995).

[22] D. Detlefs, P. Martin, M. Moir, G. Steele, Lock-free reference counting, in: Symposium on Principles of Distributed Computing, ACM, 2001, pp. 190–199.

[23] T. Harris, A pragmatic implementation of non-blocking linked-lists, in: International Conference on Distributed Computing, Springer, 2001, p. 300–314.

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

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

[26] M. Herlihy, V. Luchangco, P. Martin, M. Moir, Nonblocking memory management support for dynamic-sized data structures, ACM Transactions on Computer Systems 23 (2) (2005) 146–196.

[27] A. Gidenstam, M. Papatriantafilou, H. Sundell, P. Tsigas, Efficient and reliable lock-free memory reclamation based on reference counting, IEEE Transactions on Parallel and Distributed Systems 20 (8) (2009) 1173–1187.
[28] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, N. Shavit, Stacktrack: An automated transactional approach to concurrent memory reclamation, in: European Conference on Computer Systems, ACM, 2014, pp. 1–14.
[29] D. Alistarh, W. M. Leiserson, A. Matveev, N. Shavit, Threadscan: Automatic and scalable memory reclamation, in: ACM symposium on Parallelism in Algorithms and Architectures, ACM, 2015, pp. 123–132.
[30] N. Cohen, E. Petrank, Efficient memory management for lock-free data structures with optimistic access, in: ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2015, p. 254–263.
[31] N. Cohen, E. Petrank, Automatic memory reclamation for lock-free data structures, in: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, 2015, p. 260–279.
[32] O. Balmau, R. Guerraoui, M. Herlihy, I. Zablotchi, Fast and robust memory reclamation for concurrent data structures, in: ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2016, p. 349–359.
[33] P. Ramalhete, A. Correia, Brief announcement: Hazard eras - non-blocking memory reclamation, in: ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2017, p. 367–369.
[34] N. Cohen, Every data structure deserves lock-free memory reclamation, ACM on Programming Languages 2 (OOPSLA) (2018) 1–24.
[35] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, M. L. Scott, Interval-based memory reclamation, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2018, p. 1–13.
[36] G. E. Blelloch, Y. Wei, Concurrent reference counting and resource management in wait-free constant time, CoRR abs/2002.07053 (2020).
[37] J. Kang, J. Jung, A marriage of pointer- and epoch-based reclamation, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2020, p. 314–328.
[38] D. Anderson, G. E. Blelloch, Y. Wei, Concurrent deferred reference counting with constant-time overhead, in: ACM SIGPLAN International Conference on Programming Language Design and Implementation, ACM, 2021, p. 526–541.
[39] A. Correia, P. Ramalhete, P. Felber, Orcgc: Automatic lock-free memory reclamation, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2021, p. 205–218.
[40] R. Nikolaev, B. Ravindran, Snapshot-free, transparent, and robust memory reclamation for lock-free data structures, in: ACM SIGPLAN International Conference on Programming Language Design and Implementation, ACM, 2021, p. 987–1002.
[41] G. Sheffi, M. Herlihy, E. Petrank, Vbr: Version based reclamation, in: ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2021, p. 443–445.
[42] A. Singh, T. Brown, A. Mashtizadeh, Nbr: Neutralization based reclamation, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2021, p. 175–190.
[43] J. Jung, J. Lee, J. Kim, J. Kang, Applying hazard pointers to more concurrent data structures, in: ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2023, p. 213–226.
[44] I. G. de Wolff, D. Anderson, G. K. Keller, A. Seletskiy, A fast wait-free solution to read-reclaim races in reference counting, in: Euro-Par: European Conference on Parallel and Distributed Processing, Part III, Springer, 2024, p. 103–118.
[45] G. Sheffi, E. Petrank, The era theorem for safe memory reclamation, in: ACM Symposium on Principles of Distributed Computing, ACM, 2023, p. 102–112.
[46] M. Areias, R. Rocha, Towards a lock-free, fixed size and persistent hash map design, in: International Symposium on Computer Architecture and High Performance Computing, IEEE Computer Society, 2017, pp. 145–152.
[47] P. Moreno, M. Areias, R. Rocha, On the implementation of memory reclamation methods in a lock-free hash trie design, Journal of Parallel and Distributed Computing 155 (2021) 1–13.
[48] M. Herlihy, J. M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.
[49] P. Moreno, R. Rocha, Releasing memory with optimistic access: A hybrid approach to memory reclamation and allocation in lock-free programs, in: ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2023, p. 177–186.
[50] J. Evans, A scalable concurrent malloc (3) implementation for freebsd, in: BSDCan Conference, 2006.
[51] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism, O'Reilly, 2007.
[52] R. Leite, R. Rocha, Lrmalloc: A modern and competitive lock-free dynamic memory allocator, in: High Performance Computing for Computational Science, Springer, 2019, pp. 230–243.