On Exploring Safe Memory Reclamation Methods with a Simplified Lock-Free Hash Map Design *

Pedro Moreno[1,2][0000−0003−3745−5845], Miguel Areias[1][0000−0003−1589−3174], and Ricardo Rocha[1][0000−0003−4502−8835]

[1] CRACS/INESC TEC, Department of Computer Science,
Faculty of Sciences, University of Porto
Rua do Campo Alegre s/n, 4169-007 Porto, Portugal
{pmoreno,miguel-areias,ricroc}@dcc.fc.up.pt
[2] Instituto de Astrofísica e Ciências do Espaço, University of Porto
CAUP, Rua das Estrelas, 4150-762 Porto, Portugal

Abstract. Lock-freedom offers significant advantages in terms of algorithm design, performance and scalability. A fundamental building block in software development is the usage of hash map data structures. This work extends a previous lock-free hash map to support a new simplified design that is able to take advantage of most state-of-the-art safe memory reclamation methods, thus outperforming the previous design.

1 Introduction

Lock-freedom is an important technique that is known to offer significant advantages in terms of algorithm design, performance and scalability, therefore improving the overall throughput of concurrent data structures. Hash maps are a very common and efficient data structure used to organize information that must be accessed frequently. Hash tries are a tree-based data structure with nearly ideal characteristics for the implementation of hash maps [2], which allows to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions.

In this work, we focus on simplifying a sophisticated implementation of a lock-free trie-based hash map, named *Lock-Free Hash Tries (LFHT)* [1]. An important disadvantage of the LFHT data structure is that it is incompatible with most state-of-the-art *safe memory reclamation (SMR)* methods [6]. In order to simplify LFHT's design, we redesigned the internal representation of leaf nodes so that collisions on hash buckets do not form a linked list, but are instead stored in specialized arrays. This design simplification avoids the previous disadvantage, allowing the new design to be compatible with most SMR methods. In particular, experimental results show that the new design is able to effectively take advantage of the *Optimistic Access* method [3,7], arguably one of the most performant SMR methods, outperforming the previous LFHT design.

* This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within projects UIDB/04434/2020, UIDP/04434/2020 and UIDB/50014/2020. DOI 10.54499/UIDB/50014/2020.

2 Simplified Lock-Free Hash Tries (SLFHT) Design

The original LFHT design 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. To deal with collisions, the leaf nodes form a linked list in the respective bucket entry until a threshold is met and, in such case, an expansion operation updates the nodes in the linked list to a new hash level H_{i+1}, i.e., instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w.

The key idea behind the new SLFHT design is to replace these collision chains with a specialized array of leaf nodes 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. We call these arrays of leaf nodes as *leaf arrays*. The insertion procedure, instead of adding a node to the chain, replaces the entire leaf array with a new one containing all the previous leaf nodes plus the new node. 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 1 shows an example of the removal of the nodes K_1 and K_2 from a hash level H_i. We show the initial state in Fig. 1(a). Then, in Fig. 1(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. 1(c), we remove K_1 by emptying the corresponding bucket entry B_k as there are no nodes left to keep in it.

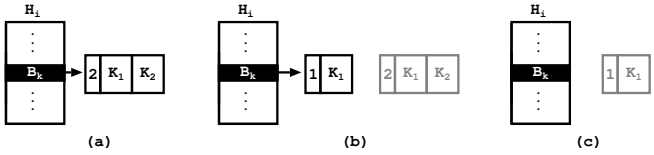


Fig. 1: Removal of nodes in a hash level using leaf arrays

3 Safe Memory Reclamation

Safe memory reclamation (SMR) 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. To reclaim the memory of a leaf array on the SLFHT design, an SMR method needs to be used to ensure that no thread possesses an old reference to it, at the consequence of triggering an use after free bug. The choice of the SMR method is therefore of greater importance. We implemented two different SMR methods for the SLFHT design. We chose the *Hazard Pointers* (HP) method [5], as it is the most commonly used one and tends to achieve good performance in data structures with low depth with tight memory bounds. As a

second option, we chose the *Optimistic Access* (OA) method that was developed by Cohen and Petrank [3], and then further improved by Moreno and Rocha [7]. The OA method is one of the most efficient memory reclamation methods while being robust and simple to implement.

4 Performance Analysis

Our experimental environment was a machine with 2 x AMD Opteron^{TM} Processor 6274 with 16 cores each and 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).

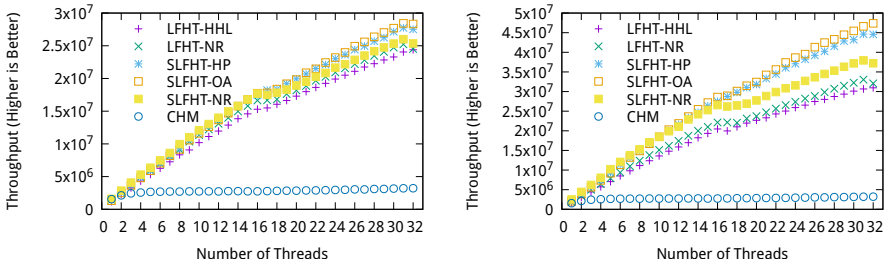
For the benchmarks, we used multiple artificial scenarios with varying ratios of insert, search and remove operations (keys were pre-inserted for the search and remove operations) and with different configurations. We ran configurations with 2^4 and 2^8 bucket entries per hash node combined with an expansion threshold on 3 and 5 nodes. All scenarios execute a total of 10^7 operations with uniformly randomized keys. Each scenario was ran 5 times. The results that follows are shown in throughput (operations per second).

For the performance analysis, we evaluate SLFHT against LFHT and the Concurrent Hash Map design (CHM) of Intel-TBB library version 2021.5.0. The scenarios in Fig. 2 and Fig. 3 specify the design and the 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 [6], the HP (Hazard Pointers) and OA (Optimistic Access) are the two SMR methods discussed 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 [7], we used the LRMalloc memory allocator [4] for all versions.

Figure 2 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 achieved with reclamation, as the number of allocations and frees are closely matched with memory reclamation. Figure 3 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, overrunning also the CHM design.

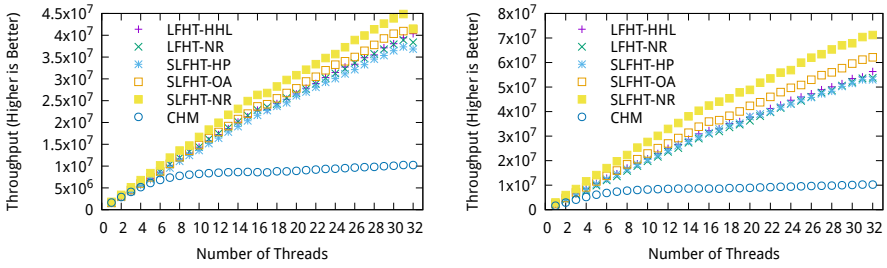
5 Conclusions

We have redesigned the LFHT data structure in order to make it compatible with most SMR methods. Its design simplicity makes it more desirable and reliable for adoption in real world applications. Experimental results show that the new SLFHT design achieves significant performance gains when compared against the old LFHT design and the CHM design supported by Intel.



(a) 2^4 bucket entries and threshold 3 (b) 2^8 bucket entries and threshold 5

Fig. 2: 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

Fig. 3: Throughput for the 90% *Searches*, 5% *Inserts* and 5% *Removes* scenario

References

1. Areias, M., Rocha, R.: Towards a Lock-Free, Fixed Size and Persistent Hash Map Design . In: International Symposium on Computer Architecture and High Performance Computing. IEEE (2017)
2. Bagwell, P.: Ideal Hash Trees. Es Grands Champs **1195** (2001)
3. Cohen, N., Petrank, E.: Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In: ACM Symposium on Parallelism in Algorithms and Architectures. ACM (2015)
4. Leite, R., Rocha, R.: LRMalloc: A Modern and Competitive Lock-Free Dynamic Memory Allocator. In: High Performance Computing for Computational Science. Springer (2019)
5. Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems **15**(6) (2004)
6. Moreno, P., Areias, M., Rocha, R.: On the implementation of memory reclamation methods in a lock-free hash trie design. Journal of Parallel and Distributed Computing **155** (2021)
7. Moreno, P., Rocha, R.: 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)