Reclaiming Memory from Lock-Free Hash Tries[⋆]

Pedro Moreno and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{pmoreno,ricroc}@dcc.fc.up.pt

Abstract. This work presents an efficient memory reclamation scheme applied to an implementation of lock-free hash tries which follows a methodology based on the concept of hazard pointers.

Keywords: Memory Reclamation · Lock-Freedom · Hazard Pointers.

1 Introduction

Nowadays, there are multiple implementations of efficient lock-free data structures but, most often, no memory reclamation method is proposed for them. Researchers tend to either declare them outside of their scope or rely upon a general purpose garbage collector [1]. However, such garbage collectors are rarely lock-free or very efficient compared to manual memory management.

The most relevant lock-free memory reclamation schemes known are based on two main methodologies: (i) *hazard pointers* [4] and (ii) *temporal order between events*, which are usually achieved by either *quiescent states* [3] or *epochs* [2, 3].

With hazard pointers, each thread is responsible for keeping the (pointers to the) elements it is visiting in globally visible memory, thus rendering them irreclaimable during the visiting period of time. This method is very efficient in terms of memory usage but requires two atomic writes for every visited element, which can lead to huge overheads. Quiescent states and epochs rely on keeping a temporal order between logical removal of elements and the instants in which a thread has no reference to any element. Both methods allow very time efficient implementations but have the disadvantage of not guaranteeing a bound on the amount of unrecovered memory, revoking the theoretical lock-freedom property of the system as no progress is guaranteed for reclamation.

Starting from the hazard pointers methodology, in this work, we present an efficient memory reclamation scheme applied to an elaborate implementation of lock-free hash tries [1], giving us the ability to exploit the hash tries' structure to achieve high efficiency, low memory bounds and even enhance the data structure itself. We believe that our approach can be extended and applied to similar tree-based data structures.

[⋆] This work was funded by the ERDF through Project 9471-RIDTI – *Reforçar a Investigação, o Desenvolvimento Tecnológico e a Inovação* – and through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-016844, and by National Funds through the FCT as part of project UID/EEA/50014/2013.

2 Hash Tries Overview

Hash tries are a tree-based data structure with nearly ideal characteristics for a hash map implementation. We based our work on the lock-free implementation of hash tries proposed by Areias and Rocha [1]. The implementation has two kinds of nodes: *hash nodes* and *leaf nodes*. The leaf nodes store key/value pairs and the hash nodes implement a hierarchy of hash levels of fixed size 2^w. To map a key/value pair (k, v) into this hierarchy, we compute the hash value h for k and then use chunks of w bits from h to index the appropriate hash level, i.e., for each hash level H_i, we use the $w * i$ least significant bits of h to index the entry in the appropriate bucket array of H_i. Figure 1 shows a small example that illustrates how the insertion and expansion of nodes is done in a hash level.

Figure 1(a) shows the initial configuration for a hash level. Each hash level H_i is formed by a bucket array of 2^w entries and by a backward reference $Prev$ to the previous hash level. B_k represents a particular bucket entry of H_i. A bucket entry stores either a reference to a hash level (initially the current hash level) 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 $K1$ on B_K and Fig. 1(c) shows the configuration after the insertion of nodes $K2$ and $K3$. A leaf node holds both a reference to a next-on-chain node and a flag with the condition of the node, which can be valid (V) or invalid (I).

When the number of valid nodes in a chain exceeds a given threshold, the corresponding bucket entry is expanded to a new hash level and the nodes in the chain are remapped in the new level, 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 expansion operation starts by inserting a new hash level H_{i+1} at the end of the chain (as shown in Fig. 1(c)) and then it moves the leaf nodes, one at a time, from H_i to H_{i+1}. Figures 1(d) and 1(e) show how node $K3$ is first mapped in H_{i+1} (bucket B_n) and then moved from H_i (bucket B_k).

Removal of keys are performed by first marking the leaf node as invalid (setting its flag to I) and then by setting the next reference of the previous valid node to the first valid node subsequent to the node being removed [1].

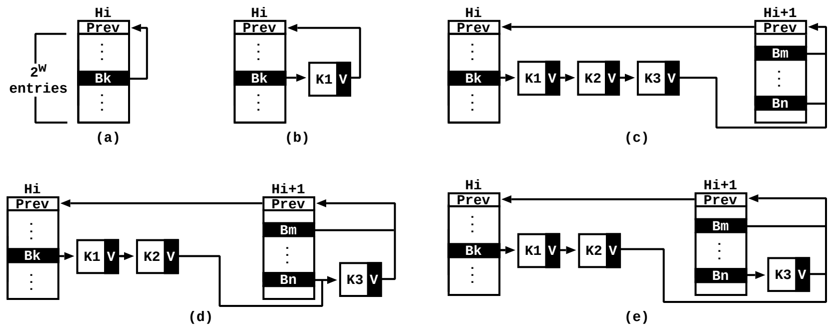

Fig. 1. Insertion and expansion of nodes in a hash level

3 Our Memory Reclamation Scheme

To reclaim memory, we need to be sure that no thread can further use such memory. A memory element M being removed can be easily made unreachable (*logical remove*) for the upcoming threads. The problem arises if a thread had accessed M before the logical remove and can still access M in the future. To *physically remove* M and reclaim its memory we need to ensure that no thread can further use M. To ensure this, we can use hazard pointers for individual elements but, as we have seen, they are very time consuming. Instead, we propose using the hazard pointers concept but as a way to record the hash bucket entry and the corresponding chain of leaf nodes where a thread is operating. This reduces the number of updates to the points where threads move between hash levels. For that, we can use the hash value h as a witness of the path a thread is traversing and the hash level l as the part of such path the thread is currently on. In what follows, h and l will be named as the *hazard hash (HH)* and the *hazard level (HL)*, respectively. To prevent unbounded memory usage while maximizing performance with this new approach, the following extensions where introduced to the previous design.

- Bucket entries were extended to include a *bit flag* indicating if the stored reference is for a next level hash node. The new flag is part of the atomic field including the reference. Whenever the flag is set, the entire atomic field becomes immutable.
- Leaf nodes were extended to include a *generation field* indicating the hash level the node was first inserted on, and a *tag* indicating the hash level the node is at the moment. The new tag is part of the atomic field including the next-on-chain node and the flag with the condition (valid/invalid) of the node. Whenever the tag is updated, the entire atomic field is updated.
- Threads now collaborate to finish the expansions in course in a path before inserting new nodes. This ensures that no more than one expansion will be occurring in a path at any given moment.
- If the number of nodes protected from memory reclamation by a thread's hazard pair $< HH, HL >$ exceeds a given threshold, an expansion operation has to be forced. This ensures that each thread T will have a bound on the number of nodes it can be preventing from being reclaimed. This solves the problem of having multiple nodes being inserted and removed in the chain T is in, without triggering the threshold limit for expansion.

With these extensions in mind, our memory reclamation scheme works as follows. When a node N is being removed by a thread T, N is first made unreachable for the upcoming threads (logical remove) and then stored on a *local reclamation list*, from which T periodically tries to reclaim nodes (physical remove). The reclamation procedure is similar to hazard pointers with the difference that each hazard pair $< HH, HL >$ protects a (logically) removed node N from memory reclamation if: (i) N has a hash h_N matching the hazard hash HH up to the hazard level HL; (ii) $generation_N \leq HL$; and (iii) $tag_N \geq HL$. Note that the tag preserves the level at which the node was invalidated and logically removed.

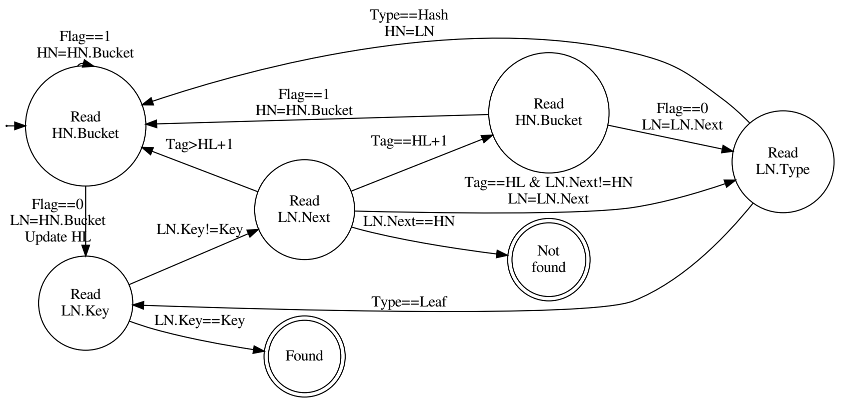

Fig. 2. Deterministic state machine representing the search for a node

Figure 2 illustrates the new set of states for traversing the hash trie structure searching for a node. A key point of our scheme is to ensure that the traversal procedure does not dereference a next-on-chain node if such node is not being protected by the current hazard pair $< HH, HL >$. When a leaf node (LN) contains a tag value equal to our current hash level plus 1 ($HL + 1$), we know that an expansion operation has started and we need to verify if the expansion is still in course by checking the flag in the bucket entry of the last hash node HN seen. A flag value of zero means that the expansion is still in course, so we can be sure that the next-on-chain reference we have read earlier is to a node that was expanded from the previous level and is thus still protected by our hazard pair. On the other hand, if a flag value of 1 is found, we can simply follow the reference in the bucket entry as a way to reach the next hash level and continue the traversal. Similarly, if we find a tag value greater than $HL+1$, we can be sure to find the next hash level in the bucket entry of the last hash node HN seen. This scheme grants us a memory bound, which depends on the number of threads, hash levels and threshold values, while requiring a number of atomic writes per operation very close to temporal order schemes (bounded by the maximum number of levels).

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 Applications and Technologies. pp. 145–152. IEEE (2017)
2. Fraser, K.: Practical Lock-Freedom. Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory (2004)
3. Hart, T.E., McKenney, P.E., Brown, A.D., Walpole, J.: Performance of Memory Reclamation for Lockless Synchronization. Journal of Parallel and Distributed Computing **67**(12), 1270–1285 (2007)
4. Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems **15**(6), 491–504 (2004)