Efficient Lock Free Binary Search Trees

*Final Project Paper for COP 6616 Multicore Programming by Dr. Damian Dechev

Fnu Tulha
Computer Science PhD student
University of Central Florida
Orlando, Florida
tulha@Knights.ucf.edu

S.M. Iqbal Morshed
Computer Science PhD Student
University of Central Florida
Orlando, Florida
imoshed@Knights.ucf.edu

***Abstract*—In this paper, we discuss our findings related to the implementation of the efficient lock free binary search tree described in [1]. We re-implement the aforementioned data structure and perform an experimental evaluation on it. In addition, we also talk about the performance of our concurrent data structure in relation to a transformed transactional data structure designed using software transactional memory and additionally use evaluation metrics to compare the performance for both versions.**

***Index Terms*—lock free synchronization, binary search tree (BST), software transactional memory, compare and swap**

I. Introduction

Lock free synchronization has been a challenging problem in the field of parallel programming for a long time. Since the use of locks reduces the efficiency of an algorithm by adding an overhead of the communication between multiple threads or processes, lock free algorithms are sought that function independently of locks. One of the challenges of ensuring lock free synchronization, as explained in [2], is due to the synchronization instructions being more expensive in terms of operations than regular memory accesses. Various lock free algorithms have been presented for different data structures including queues [4], stacks [3] as well as binary search trees(BSTs) [1].

At the time of publishing of [1] there were various problems with the lock free algorithms for BST's. Some used multi word compare and swap operation (MCAS) as in [6] to solve the issue of multiple threads contending at a node for either deletion, addition or contains. But due to the portability issue of the MCAS ,this is not an effective solution for platforms incapable of supporting it. Other versions of the algorithm used the concept of flagging an entire node for deletion and then used CAS to make changes to it atomically. This approach works but it is very inefficient because when a node is occupied for deletion, all other operations' traversing on the said node keep spinning until the remove operation either succeeds or fails. In the case, where the contention gets large, the aforementioned approach would cease to be effective. Other solutions proposed allow BST's to store the external and internal nodes separately by the names data and null nodes respectively. The problem with this solution is the use of extra memory to store these different representation leading to a memory inefficient solution.

It is noteworthy that the performance bottleneck of any lock free BST implementation is going to depend on how efficiently the remove operation is implemented. The different cases for the removal of a node make it difficult to design a strategy that can be effective in a high contention environment. Designing the add and contains operation is less involved due to the fact that addition of a node always takes place at the leaf nodes of a BST.

In this paper, we provide a re-implementation of the lock free algorithm for BST's presented by Chaterjee et al. [1] and discuss the performance of our version of the data structure even though the actual paper does not provide an implementation and experimental evaluation. After presenting our interpretation of the lock free algorithm and after due consideration of various factors that affect the performance, we transform this data structure into a transactional data structure by using RSTM [6] which is an implementation of software transactional memory for C++.

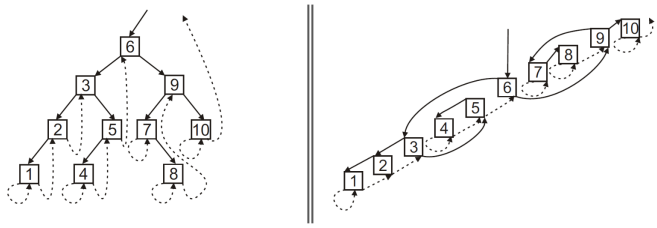

Fig. 1. A threaded BST can be treated as a linked list of nodes where each node's predecessor, successor, parent and children can be effectively accessed [1].

We will finally conclude our paper by comparing the performance for both versions of our data structure, concurrent and transactional and will present our experiment results that were performed for different combinations of read and write operations and for different transaction sizes. A discussion of the strengths and weaknesses of our version of the algorithm versus the authors[1] will give the reader a better idea of the challenges of designing lock free algorithms, their scalability and potential for further research.

II. Modified Binary Search Trees

A. Threaded and Order Links

We will begin with a brief description of a sequential BST and will proceed by explaining how it is transformed into a concurrent data structure by introducing new features.

A threaded link is introduced as in [7], where each node if it does not have a left child is self threaded whereas i it's right child is null, it is right threaded so that the link is connected to its successor. This feature allows a node to have a 'threaded' access to its successor and will prove to be very useful as we will observe in later sections. As visible in Fig. 1, the threaded links exist at the leaf nodes because they have neither a left or a right child. For the node containing 5, we observe a threaded right link to its successor, which in this case is the root of the BST. The left and right children of an external node, in our representation, will be set to null. This will help us perform checks on the threaded links at a leaf node so as to terminate our traversal or allow a thread to successfully add a node to the BST. It is also interesting to point out that a traversal through this modified BST will remain unaffected by a concurrent remove operation.

An order link is defined as a threaded link that is incoming into a node. For instance, in Fig. 1, the order link for the leaf nodes is the threaded link incoming into them. For leaves, the order link emanates from themselves but for other threads it may be coming from other nodes in the BST. Other important features include the use of backlinks in the BST due to which each node has access to its parent in case there is a point of failure encountered during the traversal process. This is perhaps the most important contribution proposed by [1], due to the fact that it leads to a much more efficient amortized time complexity for the removal operation where multiple threads are contending for deletion at the same node as will be discussed in details in the correctness and complexity section of this paper.

The representation of a node in our code is basically done using a structure that consists of the child pointers and the value. Another important thing to mention is that each link is represented by a 4 tuple, that is represented by $(Reference, flagged, marked, threaded)$. Reference is the location in memory that points to where the link of a node is stored, flagged and marked are bits that indicate whether the link is marked or flagged for deletion. Finally, the threaded bit tells us whether the link in question is threaded or not. For example, $(R, 0, 0, 1)$ would correspond to a link that is stored at a memory location R and is threaded but not marked or flagged.

B. Category Nodes

Another importance feature of the new data structure is the specification of special nodes denoted by three different categories namely category 1, 2 and 3. This specificiation of nodes in the BST is significant to deal with the different cases of a remove operation in the BST. The different category nodes arre defined in Fig. 2 as the following:

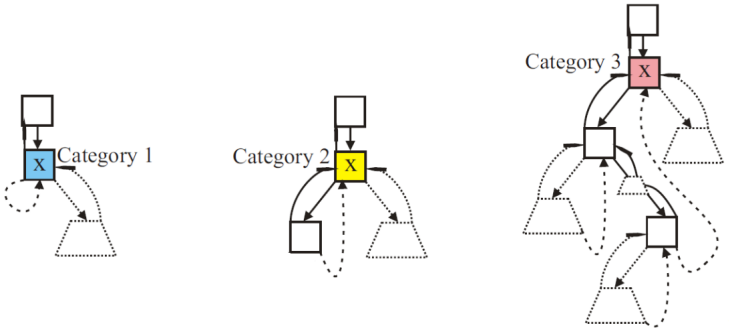

Fig. 2. The different category nodes shown according to the source of the order link.(a) Order link source: Same node. (b) Order link source: left child (c) Order link source: right most child of left subtree or predecessor [1].

- *Category One*: These nodes consist of an order link that emanates from themselves. For node x in Fig 2(c), the order link is incoming into the node from itself; this self loop leads to it being referred to as a category 1 node. Removal for a category 1 node will be trivial since only its right child or subtree need to be connect to its parent for deletion to be valid.
- *Category Two*: For this category of nodes, the order link will emanate from the left child as shown in Fig. 2(b). For the removal of this node, we will first change the parent link to now point the x's left and right children after which the order link will be updated to the child or subtree that x was pointing to.
- *Category Three*: A category 3 node is one which has an order link that has a source in its predecessor; this node would be the right most child of the left subtree. In Fig. 2(c), we can observe the node x connected to its predecessor via an order link.

It is interesting to observe that for adding a node to (b) in Fig. 1, we now only need to change only one pointer. However for removing a node, we will have to change a maximum of 4 pointers depending on which category node we are attempting to remove as discussed above. The resulting BST will now be converted to a new form that consists of threaded links, category nodes as well as back links. This transformed data structure that was implemented by us can be observed in Fig. 3.

III. Our Implementation

We implement our version of the lock free BST algorithm using C++11 using libraries such as atomic and pthreads. The actual algorithm details are referred to in [1] and in this section we will describe the difference in the implementation between the author's version and ours. The goal was to implement a Set Abstract Data Type (ADT) capable of performing *ADD, REMOVE and CONTAINS* operations on a BST concurrently without the use of synchronization techniques such as locks and monitors.

A. CONTAINS Operation

For all these set operations, it is understood that a predecessor query will have to be performed to locate a key k which lies between k_i and k_j meaning that it can either be greater than key or less than it and vice versa for k. Contains is used to determine precisely this; the function starts starts from a set of nodes based on current and previous key and traverses the BST.

k can either be greater, less or equal to the current key. The return value depends on the case we encounter and if k does not equal the current key, then it means that the interval we seek is related to the outgoing link from the current node x in the direction of the return value. This ensures that the last visited link in the traversal is returned so the locate

B. ADD Operation

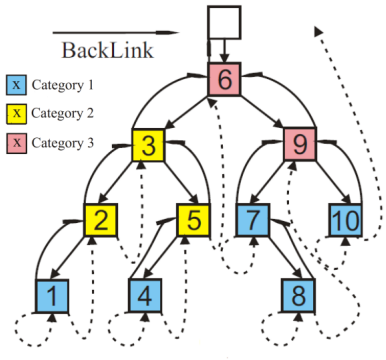

Fig. 3. The modified BST after inluding the category nodes, backlinks and threaded format [1].

To add a new node with a key, k to the current data structure, we will first call the locate function. If it succeeds (returns 2), then we dont need to add the node as it is already present. If it returns either 0 or 1 then we have to create a new node with key k. We know that when this node is initialized, both its children are threaded. The left child is threaded to itself as shown in figure 1(a) and the right child is going to take the value of the link it has to point to eventually. Using the compare and swap operation, the right link will be atomically connected to the relevant position in one step. If the compare and swap fails, then it is checked whether the target link is flagged, marked or if another add operation has successfully taken place for a similar node. If it is marked or flagged, then the nodes addition to the BST has failed and this will be retried after the new link has been identified due to a concurrent operation taking place.

C. REMOVE Operation

The *REMOVE* operation is more involved than the others and as discussed in the previous section, forms the bottleneck of our any concurrent implementation of a lock free BST. The remove operation is carried out in a series of 7 steps that will now be explained.

- *STEP I*: We begin the removal of a node by flagging the order link of a node. As explained in an earlier section, the order link points to a node itself for category 1, left child for category 2 and the successor for category 3. This step is the most important one since it allows us to eliminate the possibility of another thread attempting to delete this first.
- *STEP II*: We proceed to this step in order to set the prelink from the node to be deleted to the order node. This step ensures the presence of a solid link between the node to be deleted and its predecessor(for category 3) thereby connecting the resulting BST after removal.
- *STEP III*: The outgoing right link is marked in this step because there is a possibility that the node could be marked for deletion by another thread and we would not want it to be shifted or removed while we are trying to remove the node that we started with.
- *STEP IV*: Here, we flag the parent link of the predecessor that is linked to the node to be deleted. This is necessary since the node to be removed might be somewhere in the middle of the tree and could have a parent that is not the root in which case we want to concurrently connect the parent link of the predecessor.
- *STEP V*: The incoming parent link is finallt flagged. This is carried out because the parent of the node will cease to remain the same and will have to be updated. When we flag it, we prevent other threads from being able to remove that node.
- *STEP VII and VIII*: In the final two steps, we perform the marking of the left link of the node to be deleted and the left link of the node that is its predecessor. These steps come later in the queue since they are the least important in the removal process.
  This orderly flagging and marking of links is carried out to ensure that each thread executing concurrently is aware of the process a link is at in the removal operation. For instance, two $threads$ A and B attempting to remove the same node will try to flag the order link of the node and only one will succeed so the failed thread B will attempt to determine the stage of the order link in the removal process. This ensures each operation is able to help other pending operations so that they can speed up the removal of a node and don't waste time spinning as explained earlier in the paper.

D. Progress Guarantee and Correctness

Here we discuss the correctness of our data structure. Our set ADT is linearizable and we can explain that by considering the success and failure of various operations that are happening concurrently.

- ADD Operation: For the add operation, the linearization point would be when multiple threads attempt to use compare and swap to replace the threaded links at the leaf nodes. In case of both success and failure, the linearization point will be where the CAS returns.

- CONTAINS Operation: For the contains operation, we are traversing the BST and running the helper function locate with the desired key k. The linearization point in this case would be where the comparison between key k and k_c is carried out; ; this naturally happens in the locate function.
- REMOVE Operation: For this operation, there can be two cases that we have to deal with. If the remove operation is successful then the linearization point will be where the CAS for swapping the flagged parent link in step V of the removal process takes place. In case of failure, we can discuss two cases. If the node is not located, then the linerization point would be the linearization point of the unsuccessful locate operation called by the unsuccessful remove. However, in case a node is located, then that means another thread has successfully removed the node so the linearization point in this case would be that of the thread that actually removed the node.

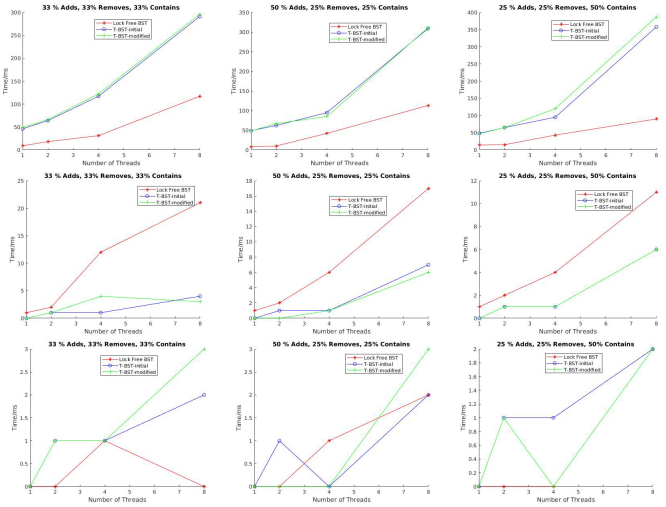

Fig. 4. Experimental Evaluation.

Our data structure is therefore linearizable and lock free. It is however, not wait free since there is not guarantee that all the operations would be successful in a bounded number of steps. The key synchronization used to design this data structure was atomic modification of multiple pointers by exploiting the use of flags at each link. This technique is similar to fine grained locking as we attempt to not flag an entire node and prevent other threads from performing useful work on it while it is being deleted. This is of course similar to the coarse grained locking idea and we would like to avoid that in the design of synchronization techniques as much as possible.

There were many differences between our implementation and the algorithm discussed in [1].Our choice of variables which were chosen to be made atomic were not the same as the ones mentioned in the variable. Since the paper provided only a theoretical model of a lock free BST and did not provide details of the implementation in a programming language, we enjoyed freedom in selecting the technique that could potentially be used to implement the algorithm.

One of the best features of the dataset are that all the operations aid each other in their respective ways. A contains enjoys full freedom in its traversal because it is able to monitor the links in its path for flags and marks. The contains operation always returns TRUE for a node if it is not physically deleted meaning it still has a parent. A node may be logically deleted during a traversal but since it still has a parent it is technically part of the BST. This data structure performs better than the counterparts proposed before [1] was published.

The challenges that we encountered during the project was mostly due to major pieces of information being omitted by the authors about the pseudocode. There were some major mistakes in the code as well that resulted in segmentation faults that were difficult to debug and resolve.

E. Experimental Results and Conclusion

Our version of the lock free algorithm was written using C++11. In Fig. 4, the red curve shows the performance of our implementation without the use of transactional memory. The y axis displays the execution time for the test case in milliseconds for 1, 2, 4 and 8 threads. The blue and the green curves display the performance for the transactional data structure: initial and modified respectively.

We have created three test case scenario. In each thread, we have executed 10,000 operations. The ratio of add, remove and contains operations in the three test case scenario were (33%, 33%, 33%), (50%, 25%, 25%) and (25%, 25%, 50%) respectively.

We used RSTM to transform our concurrent data structure to a transactional data structure. In this section we will present the results of our findings and discuss them as well.

It can be observed in the given figures that there are three benchmarks that were performed. We name our initial transactional data structure T-BST-initial and after modifying certain parameters such as the aligned memory allocated to the transactional data structure we reevaluate the results on T-BST-modified. We ran the experiment for varying transaction sizes including 1, 50000 and 500000 and observed the performance of our data structure to be consistent with transaction size even for a different mix of operations. For instance, the first row shows operations with transaction sizes 500000 and in this case our lock free data structure out performs the transactional memory. We predict that this occurs due to overhead in memory monitoring when there is a lot of contention present on the data structure. With the increase in the number of threads the execution time increases approximately exponentially for the transactional memory but is linear for the lock free data structure.

The reason for this is that we are only testing from threads 1 to 8 in jumps of 2 which causes the trend to appear linear. However, if the tests were run for 32 threads as was the case in the assignment, the trend would be observed to be more irregular.

F. Conclusion

In conclusion, the data structure we designed outperforms the transaction data structure implemented using RSTM for a large transaction size. We observed a better performance of the transactional data structure for smaller transactions. That said, there is obviously room for improvement for both versions of our data structures. An interesting project extension could the comparison of RSTM and GCC transactional memory and a comparison of benchmarks to ascertain which transactional memory implementation is better for binary search trees.

REFERENCES

[1] 1. Chatterjee, Efficient Lock Free Binary Search Trees
[2] 2. https://homes.cs.washington.edu/ lamarca/pubs/lockfree-podc94.pdf
[3] Lock Free Stack https://people.csail.mit.edu/shanir/publications/LockFree.pdf
[4] Lock Free queues http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.8674rep=rep1type=pdf
[5] A Highly-Configurable C++ Software Transactional Memory (STM) Library, https://code.google.com/archive/p/rstm/
[6] K. Fraser. Practical lock-freedom. PhD thesis, Cambridge University, Computer Laboratory, 2004.
[7] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. Communications of the ACM, 3(4):195204, 1960.