Aryan Soman – asoman

Jerry Xu – jerryx

15418 Final Project Report

Concurrent AVL Trees

Summary

We implemented three different AVL trees: one sequential tree for benchmarking, one concurrent tree with a fine-grained per-node locking scheme, and one with a lock-free concurrency algorithm. We wrote correctness tests for all of our implementations. We also measured their performance on the PSC Bridges-2 machine while varying the number of threads, the ratio of read to write operations, and the cardinality of the set of keys.

Background

Binary search trees are one of the most important structures for storing data that obeys a total order. We focused on the insert, search, and remove operations for this project. Various tree rebalancing schemes, such as red-black, splay, and AVL (the focus of this project), exist to ensure that the height of the tree is logarithmic in the number of nodes; this underlies the efficiency of the BST operations relative to a linear structure like a linked list. We define the height of a node as

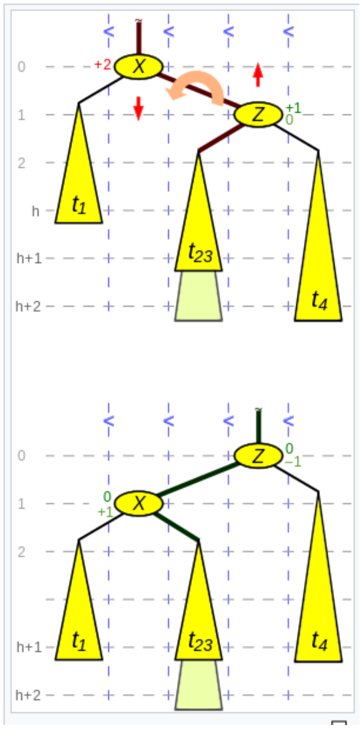
$$h(\text{node}) = 1 + \max(h(\text{node->left}), h(\text{node->right})), \text{ where } h(\text{NULL}) = 0.$$

The AVL scheme ensures that at the end of each operation, for any non-NULL node,

$$|h(\text{node->left}) - h(\text{node->right})| < 2.$$

It is proven that this invariant ensures that the height of the tree (defined as h(root)) is O(log n), where n is the number of nodes in the tree.

The standard BST write operations (insert and remove) can cause the AVL invariant to be violated. Therefore, AVL write operations have a rebalancing phase after insertion or deletion of a node, which causes the tree to restore the AVL invariants before the start of the next operation. This rebalancing phase consists of a series of tree rotations. There are four types of rotations; the type we apply on a node depends on the relative heights of the node and its children. The initial rebalance can be arbitrarily high (close to the root) on the tree relative to the modification point. Moreover, rotations can cause imbalance to propagate up the tree, so we continue applying rotations to higher and higher nodes until the imbalance is eliminated. Each rotation is constant time and the total number of rotations is at most the height of the tree, O(log n). Therefore, the rebalance phase is also O(log n), ensuring that each operation is still O(log n) time overall.



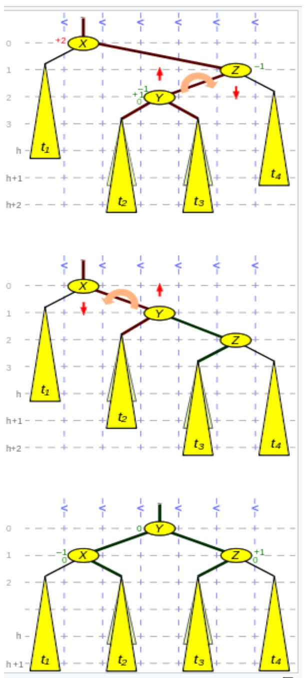


Fig. 3: Double rotation *rotate_RightLeft(X,Z)* = *rotate_Right* around *Z* followed by *rotate_Left* around *X*

Fig. 1: Diagrams of single and double rotation. https://en.wikipedia.org/wiki/AVL_tree

Having a thread-safe AVL tree would be useful so that multiple users could access the tree at the same time, which is a very common case for real-world data stores. Beyond simple coarse-grained locking schemes, per-node locking schemes and lock-free algorithms exist for AVL trees. The idea is that different threads can operate on non-intersecting parts of the tree at the same time.

These concurrent algorithms usually assume the existence of a maintenance thread—a thread that is responsible for operations like balancing the tree and pruning deleted nodes—while all the other threads do the actual reads and writes. As a result, relative to coarse-grained locking, which operates sequentially, we would expect the optimal speedup from the fine-grained locking and lock-free algorithms to be totalThreads-1 for a large enough tree, since that’s how many operations can take place at once when the modifications don’t collide.

Many of these algorithms, including the fine-grained locking and lock-free implementations we cover here, maintain a “relaxed-balance” invariant to facilitate performance. This means that the tree may not satisfy the traditional AVL balance invariant after each modification. It is then the responsibility of the maintenance thread to rebalance the relaxed tree into an AVL tree in finite time.

Approach

Implementation Tools

We implemented all three AVL trees and all tests in C++. We wrote multithreading in our tests using the pthread interface, used std::mutex as a lock in our locking implementation, and used the compare_exchange_strong function (member of std::atomic<T>) for compare-and-swap functionality in the lock-free implementation. We initially tested our code on the 8-core GHC machines. For our final performance results, we ran our code on the PSC Bridges-2 machine to be able to test with a wider range of thread counts.

Coarse-Grained Locking Algorithm

The simplest method to make an AVL tree thread-safe is for a thread to lock the whole tree while it does an operation. However, this allows one thread to access the tree at a time, making it essentially sequential in terms of operational throughput. As a baseline, we created a sequential AVL tree using the traditional algorithm from Wikipedia which we tested using one thread for each set of parameters (more on this later).

Fine-Grained Locking Algorithm

There is a problem with any fine-grained locking scheme that tries to adhere to the AVL invariants exactly. With the AVL rebalancing scheme, an imbalance in left and right height can occur arbitrarily high on the tree, as shown in Fig 2.

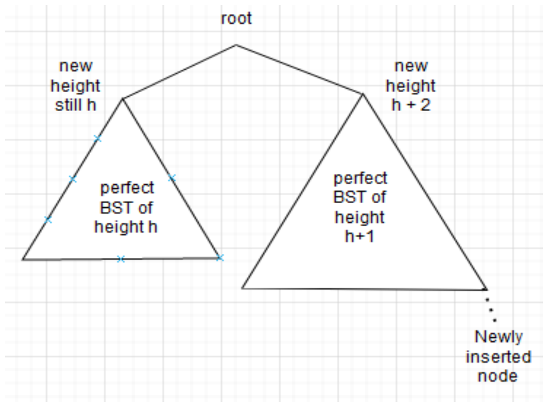


Fig 2: an AVL tree with a newly inserted node where the lowest instance of imbalance is as high as the root

This means that a thread in such a fine-grained locking scheme must still maintain locks on the whole path it took to the modification point to ensure that the subsequent rebalancing is thread-safe. Particularly, the root would be locked, which locks every other thread out of the tree, making a naive fine-grained locking scheme no better than the coarse-grained lock.

To remedy this, Nurmi, Soisalon-Soininen, and Wood (1996) proposed a relaxed-balance scheme, where the tree adheres to a set of “relaxed invariants”. The advantage is that after a BST write operation to the tree that violates the relaxed invariants, it is possible to modify only a constant number of nodes around the write point to adhere to the relaxed invariants again (rather than having the modification be arbitrarily high in the tree). This means a writing thread needs to lock only a constant number of nodes at a time.

The relaxed invariants are the same as the traditional invariants except that they use “relaxed height” instead of height. Each node in the tree has a “tag” field that controls how much that node contributes to the relaxed height: a node will contribute 1 + tag to the relaxed height of any path containing it, whereas every node only contributes 1 to the true height of a path containing it. Rigorously, the relaxed height of a node is defined as

rh(node) = 1 + node->tag + max(rh(node->left), rh(node->right)) where rh(NULL) = 0.

Then the relaxed balance invariant is that for every non-NULL node,

|rh(node->left) - rh(node->right)| < 2.

We can modify the tags of a constant number of nodes during insertion and deletion to ensure that every node has the same relaxed height it did before the operation. For example, when inserting a new node, we decrement the tag of its parent to offset the increase in true height. Again, this ensures that insert/search/write threads only have to lock a constant number of nodes at a time.

However, note that obeying the relaxed invariants does not ensure that the tree is O(log n) in height (we could just modify tags forever while making a very unbalanced BST). Therefore, Nurmi et al. make use of a maintenance thread responsible for modifying the tree to get closer to a traditional AVL tree. This rebalancing thread operates independently from the threads that are reading and writing from the tree, and also will lock a constant section of nodes at a time. Nurmi et al. define a function

unbalance(node) = |node->tag| * (size of tree - size of subtree rooted at node)

The total unbalance of the tree is defined as the sum of the unbalances of all nodes. Note that if the total unbalance is 0, then all the nodes must have unbalances and tags 0. When all the nodes have tag 0, the relaxed height of each node is the same as the true height. Therefore, obeying the relaxed invariants also gives the traditional invariants.

The rebalancing thread repeatedly runs a rebalance operation that only modifies a constant set of nodes at a time through a combination of rotations and tag manipulation. Nurmi et al. proved that every time this function is run, the total unbalance of the tree decreases. Since this unbalance is a nonnegative integer and zero unbalance means obeying the traditional invariants, for any relaxed-balance AVL tree, after a finite sequence of rebalance operations, the relaxed-balance AVL tree will obey the traditional AVL invariants.

The second notable difference in this tree is that it is an external BST, which means that the key set is stored in the leaves of the tree, while the internal nodes serve as routing nodes. This simplifies the remove operation in particular, as when removing a key we don't have to worry about placing the subtree underneath it coherently. The tradeoff is that the number of nodes can be twice as large as in an internal BST.

Lock-Free Algorithm

Like virtually all lock-free data structures, the lock-free relaxed AVL tree described by Manish Singh, Lindsay Groves, and Alex Potanin in a 2020 paper relies heavily on the fundamental compare-and-swap (CAS) primitive. Each node in the tree has a pointer to an "operation" struct that describes the operation that is queued to take place at that node—the operation could be of an insertion type or a rotation type (node removal is handled differently). If a thread wants to insert at a certain node, it must first populate the operation field of the node with the operation's details using CAS. Any thread (including the thread that first populated the operation struct) that tries to modify this node will see this operation struct and then carry out the insert operation before it can carry out the operation it originally wanted to at this node.

Similar to how the fine-grained locking tree had relaxed balancing, with a maintenance thread responsible for balancing the tree, the lock-free tree does this as well (we will cover this soon). However, it also does relaxed removal in the following fashion. Each node has a “deleted” field, and if a node is to be removed, after ensuring that no insert operation must be performed through a check of the operation struct, the calling thread simply sets the “deleted” field to true. The maintenance thread is responsible for actually pruning deleted nodes from the tree, which it does only if a deleted node has less than two children, by replacing the deleted node with the alive child (if one exists) using a CAS on the pointers of the parent.

To find such nodes, the maintenance thread repeatedly runs a DFS of the tree. Update of the node heights is also carried out by the maintenance thread during this DFS. If during the traversal the maintenance thread observes a violation of AVL invariants at a node, it initiates a rotation operation. This is difficult because we must ensure that the same thread will operate on three nodes at once in a lock-free setting. To solve the issue, the rotate operation struct has a field nodeGrabbed that shows if a particular node is “owned” by that operation. Before initiating the actual rotation, the thread trying to rotate attempts to change this field nodeGrabbed from the UNDECIDED state to the GRABBED state (using CAS) for all of the nodes involved in the rotation. If the operation struct at any node is nonempty (i.e., there is already an operation queued there), it finishes that operation before continuing with the rotation.

Another interesting thing about the lock-free algorithm is that search actually utilizes no synchronization operations. Because we only insert at the bottom of a path, prune a deleted node if it has less than two children (not having to rearrange the nodes much), and rotate at a node by creating a copy of its child (see Fig. 3), any searching thread is entirely oblivious to modifications of the actual tree.

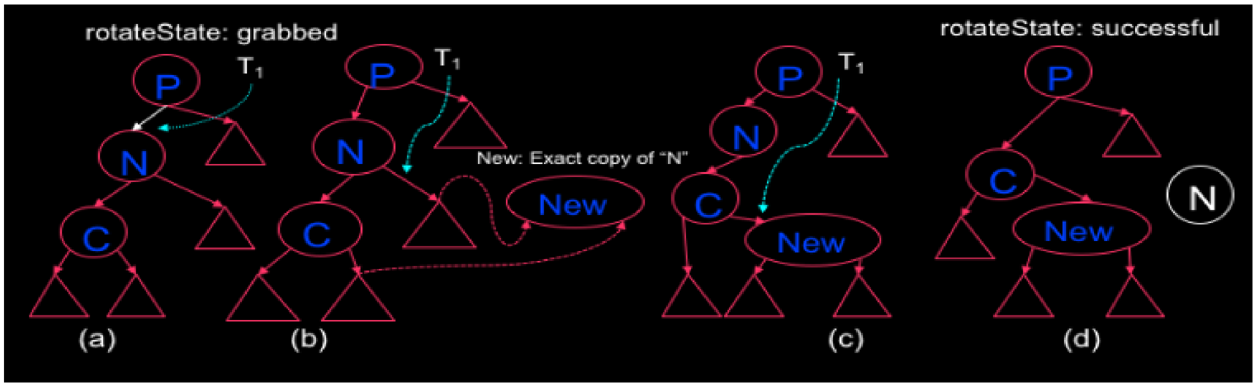


Fig 3: Rotating right at N entails making a copy of N, inserting it in the tree, and removing the old N. This ensures that a searching thread (T1) never actually sees this rotation happen. In fact, search requires no synchronization whatsoever. From Singh et al. (2020).

Correctness Testing

For each tree, we implemented both single-threaded and multithreaded tests (for the non-sequential trees). Tests consisted of a series of insert/search/remove operations being done on the tree (if we knew whether an element must have been in the tree before an operation, we also tested the result of this function call) followed by calls to search and/or the thread-unsafe getElements function to ensure that the tree had the correct elements in it. The semantics of performance testing are covered in the “Results” section. Unfortunately, our rebalance function in the lock-free tree is buggy, so we don’t have correctness tests for the rebalance behavior, reducing the lock-free relaxed AVL tree to a lock-free vanilla BST. To run our correctness tests, one can run ``make`` and then ``./CoarseTest.exe``, ``./LockFull.exe`` and ``LockFree.exe`` for the sequential, locking, and lock-free tests respectively.

Citing Algorithms and Code

The algorithm for the sequential AVL tree was translated into C++ from the Wikipedia pages for AVL trees and binary search trees. The fine-grained locking algorithm was adapted from the

1996 paper by Nurmi, Soislon-Soininen, and Wood. They generally described their algorithm through figures and gave a rough overview of how to manage locking, so we were responsible for the logic of our C++ code for this tree. The lock-free algorithm was heavily adapted from pseudocode in the 2020 paper from Singh, Groves, and Potanin. However, there were quite a few portions of the algorithm that they left empty, such as the rebalance function and parts of the rotation helper function, which we implemented ourselves based on their description. (The empty parts were detailed in a technical report which was never actually uploaded, which we confirmed with one of the authors.) All tests were written solely by us.

Results

Varying Conditions

We tested our trees' performance by measuring how long it took to do 2^22 operations (large enough to limit the impact of thread creation overhead) in a variety of conditions. In particular, we varied:

- the cardinality of the key set *k* in [2^{15}, 2^{17}, 2^{19}]
- the ratio of read to write operations *r* in [0.2, 0.4, 0.6] (with there being an equal proportion of insert and delete operations)
- and the number of non-maintenance threads *t* in [1, 4, 16, 32, 64] (for the concurrent trees, only running with *t = 1* for the sequential tree).

The motivation for varying these three parameters is as follows: We vary *t* because we want to be able to see how much our trees can use parallelism. Varying *k* is important because in a situation with a larger key set, the tree itself can grow much larger, which also enables greater parallelism because threads can only do useful work on groups of nodes that don't intersect. However, a larger key set also means that the branches of the tree can grow much longer, so each individual operation is slower. Varying *r* is important because read operations will not cause/increase imbalance while write operations will, which causes a difference in total amount of work needed and expected height of a branch for a subsequent operation in the relaxed-balance case.

Setup for Performance Testing

Before running a test on a particular set of parameters, we ran *k/2* insert operations with randomly generated keys from [0, *k*] to ensure that our performance test would measure steady-state behavior of the tree (since we are sampling keys with replacement, this means that the tree would start with ~*0.39k* unique keys). Then, we created the operations whose throughput we would measure. Each operation's key was generated randomly from the key set, and the type of the operation was also generated randomly from a distribution according to the read-write ratio: *r/2* insert, *r/2* remove, *1-r* search. In total, we were running *t+1* threads; *t* of which did these operations and 1 maintenance thread. We assigned operations to threads using a static interleaved assignment—it is true that insert and remove take longer than search, but because of the high operation count and the random operation generation, there wouldn't be an imbalance between the threads overall. We measured performance using the time measurement high_resolution_clock from the std::chrono interface. We tested on PSC Bridges-2 because that enabled us to measure performance on 64 threads (the GHC machines only have 8 cores, so a thread count of 64 would be much too large).

Graphs

In Fig.4, for each cardinality/read-write-ratio combination, we display a graph of operation throughput vs. thread count, where the horizontal axis is the number of threads and the vertical axis is the tree's throughput in log(millions of operations per second). Note that the graphs have a logscale throughput axis because the difference between the lock-free and lock-full numbers was extremely high. These numbers are from the PSC Bridges-2 machine. The yellow line shows the lock-free performance, the red shows the lock-full performance, and the horizontal blue line is the sequential tree benchmark on a single thread. Again, note that the lock-free rebalance is buggy, so these lock-free numbers are without rebalance.

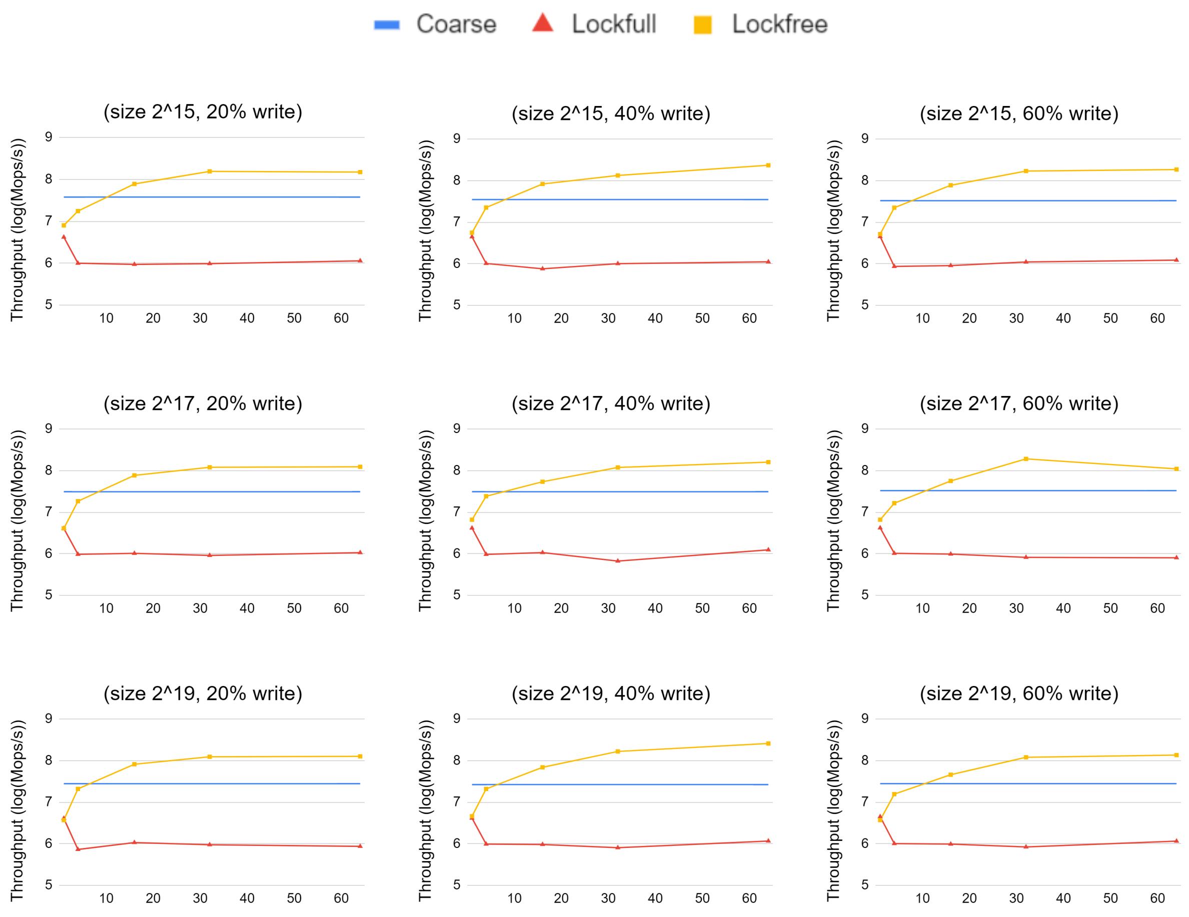
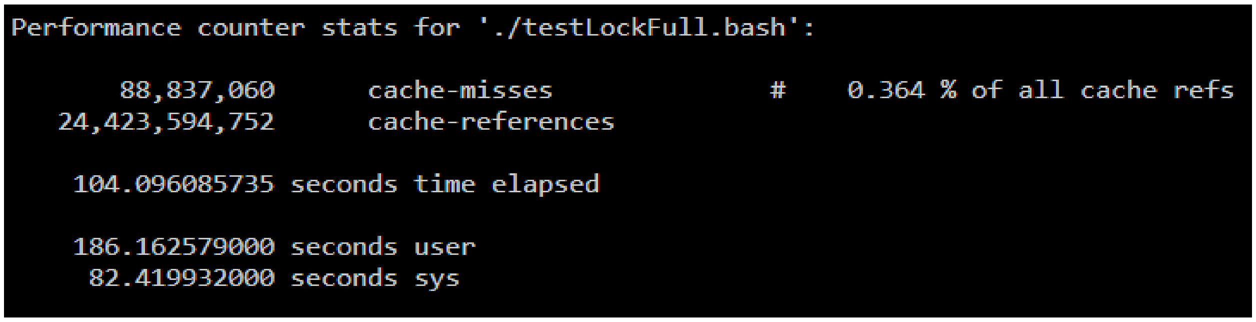


Fig 4: Graphs of throughput for the trees under various conditions.

Profiling and Interpreting Results

In general, we observed no significant differences in the results when varying the parameters. In all cases, the lock-free tree exhibits the best performance. In terms of its gains from parallelism, the speedup relative to the thread count decreases as the number of threads increases, which is common in concurrent data structures as the more threads there are, the more they must synchronize. That the lock-free tree is so fast even without rebalancing is unsurprising: it is well-documented that lock-free algorithms are often faster than their lock-full counterparts.

However, the main disappointment is the underperformance of the lock-full implementation relative even to the baseline. And not only are we not getting near the sequential baseline, but we are also seeing negative returns from multi-threading. Initially we thought that there might be a lot of contention for the locks upon release leading to cache line invalidations, but upon profiling lock-full performance tests with perf stat (Fig. 5), we discovered that the cache miss percentage was well under 1 percent. Most of the cycles were spent in the rebalance function adding to and removing from the DFS stack, but this is expected given that one thread is just rebalancing while the other threads are doing a combination of insert, remove, and search. Generally, insert is the one of these three functions that is the slowest, probably because it is the only one that allocates new memory. However, this unfortunately doesn't give us any information about why there is no lock-full speedup. We suspect at this point that it is due to the lock at the root being a bottleneck since it will be the most contended-for lock by far, but the program spends only ~1 percent of cycles waiting for a lock, so this is just speculation.

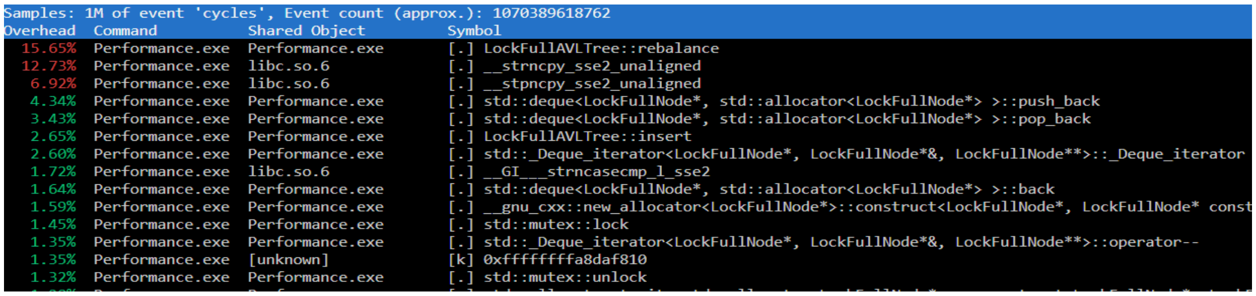


```
Performance counter stats for './testLockFull.bash':

       88,837,060      cache-misses              #    0.364 % of all cache refs
   24,423,594,752      cache-references

    104.096085735 seconds time elapsed

    186.162579000 seconds user
     82.419932000 seconds sys
```



```
Samples: 1M of event 'cycles', Event count (approx.): 1070389618762
Overhead  Command          Shared Object          Symbol
  15.65%  Performance.exe  Performance.exe        [.] LockFullAVLTree::rebalance
  12.73%  Performance.exe  libc.so.6              [.] __strncpy_sse2_unaligned
   6.92%  Performance.exe  libc.so.6              [.] __stpncpy_sse2_unaligned
   4.34%  Performance.exe  Performance.exe        [.] std::deque<LockFullNode*, std::allocator<LockFullNode*> >::push_back
   3.43%  Performance.exe  Performance.exe        [.] std::deque<LockFullNode*, std::allocator<LockFullNode*> >::pop_back
   2.65%  Performance.exe  Performance.exe        [.] LockFullAVLTree::insert
   2.60%  Performance.exe  Performance.exe        [.] std::_Deque_iterator<LockFullNode*, LockFullNode*&, LockFullNode**>::_Deque_iterator
   1.72%  Performance.exe  libc.so.6              [.] __GI___strncasecmp_l_sse2
   1.64%  Performance.exe  Performance.exe        [.] std::deque<LockFullNode*, std::allocator<LockFullNode*> >::back
   1.59%  Performance.exe  Performance.exe        [.] __gnu_cxx::new_allocator<LockFullNode*>::construct<LockFullNode*, LockFullNode* const
   1.45%  Performance.exe  Performance.exe        [.] std::mutex::lock
   1.35%  Performance.exe  Performance.exe        [.] std::_Deque_iterator<LockFullNode*, LockFullNode*&, LockFullNode**>::operator--
   1.35%  Performance.exe  [unknown]              [k] 0xffffffffa8daf810
   1.32%  Performance.exe  Performance.exe        [.] std::mutex::unlock
```

Fig. 5: Output of perf-stat/perf-record when counting cache miss percentage/number of cycles

Reproduction of Data

To reproduce our performance tests, one can run ``make``, then ``./Performance.exe treeType *r* keysetSize *t* rebalance numOps`` where treeType 0, 1, 2 corresponds to lock-full, lock-free, and sequential; percentWrite is the percentage of write operations (1 to 99); keysetSize is lg(*k*); rebalance is 0 or 1; and numOps is lg(number of total operations). To automate our tests, we also created some bash scripts that loop through these parameters and output the throughput values to a text file.

References

Here are the resources we made direct use of:

Nurmi, Otto, et al. “Relaxed AVL trees, main-memory databases and concurrency.” 1996.

Singh, Manish, et al. “A Relaxed Balanced Non-Blocking Binary Search Tree.” 2020.

Wikipedia page for AVL Tree: https://en.wikipedia.org/wiki/AVL_tree

Wikipedia page for Binary Search Tree: https://en.wikipedia.org/wiki/Binary_search_tree

Here are some other papers we looked at, but didn’t directly use:

Brown, Trevor, et al. “A General Technique for Non-Blocking Trees.” 2014.

Natarajan and Mittal. “Fast Concurrent Lock-free Binary Search Trees.” 2014.

Ramachandran and Mittal. “A Fast Lock-free Internal Binary Search Tree.” 2015.

Sueki, Sachiko. “Lock-free Self-adjusting Binary Search Tree." 2017.

Work Breakdown

Writings:	Project Proposal	Aryan + Jerry (edits)
	Project Milestone Report	Aryan + Jerry (edits)
	Project Final Report	Aryan + Jerry (edits + graphs)
	Project Poster	Aryan + Jerry
Source Code and Correctness Tests:	Concurrent AVL Tree superclass	Aryan
	Sequential AVL Tree	Aryan
	Sequential Correctness Tests	Aryan
	Fine-Grained Locking AVL Tree:	
	search	Aryan
	insert	Aryan
	remove	Aryan
	rebalance	Aryan + Jerry
	unbalance	Aryan
	printTree (debugging)	Jerry
	getElements (debugging)	Aryan
	correctness tests	Aryan
	general debugging	Aryan + Jerry
	Lock-Free AVL Tree:	
	search	Jerry
	insert	Jerry
	remove	Jerry
	seek	Jerry
	help	Aryan
	helpInsert	Aryan
	helpMarked	Aryan
	helpRotate	Aryan
	left/rightRotate	Aryan
	rebalance (buggy)	Aryan

	getElements (debugging)	Aryan
	correctness tests	Aryan
	general debugging	Aryan + Jerry
Performance Testing:	Writing performance test that is given params	Aryan
	Bash scripts looping through test parameters	Jerry
	Running performance tests on PSC-2	Jerry
	Creating graphs	Jerry
	LockFull performance investigation	Aryan

Distribution of credit:

75% Aryan

25% Jerry