A Lightweight Lock-free Concurrent Binary Search Tree (BST) Final Report
Apurva Patel (apurvapa), Harvey Ko (hungyuk)

Github Repo: https://github.com/a-patel1996/15618-project

Summary

We implemented 3 variations of a binary search tree (BST), coarse-grained, fine-grained & lock-free, and compared their performance for various workloads on the GHC machine. The lock-free implementation outperforms the other implementations.

Background

A BST is a hierarchical data structure used for storing data in a sorted manner. They are typically used in applications such as databases as it provides efficient lookup, insertion and deletion. These methods operate on nodes. Each node has at most two children referred to as the left and right child. A node contains a key that is used to identify it. It can also contain additional data related to the key. We have chosen to omit data from nodes for simplicity. There are two representations of a BST: internal and external. An internal representation is one in which data is stored in both internal and leaf nodes. An external representation is one in which data is stored in leaf nodes only and internal nodes are used for routing purposes only. The former has a smaller memory footprint since it contains about half the number of nodes than the latter while the latter greatly simplifies search, insert and delete operations on the tree in a concurrent environment.

Many applications today are multi-threaded to improve efficiency and take advantage of hardware. This necessitates augmenting data structures and algorithms to support concurrency. This is typically achieved using locks such as mutexes. Locks help to ensure that the data structure is not clobbered, causing data loss or incorrect data to be read, when multiple threads are trying to operate on it. Coarse-grained locking simply helps to serialize operations from multiple threads such that they follow some arbitrary ordering. Fine-grained locking allows multiple threads to operate on potentially different and non-overlapping parts of the tree, thus allowing multiple operations to occur in parallel. In the context of BSTs, there are dependencies with the parent and child nodes of a node being deleted or of a node at which a new node is being inserted. While locks help to prevent one thread's operations from interfering with another, it introduces contention and delay by blocking threads from making forward progress in specific scenarios. Hence, another paradigm for supporting concurrency is making data structures lock-free. This involves using special hardware supported instructions to achieve the same effect of providing atomicity and achieving a greater degree of parallelism. Although more complex to implement than a lock-based version, lock-free algorithms provide a stronger progress guarantee than blocking algorithms.

Since the lock-free implementation adheres to an external representation, we wanted to follow suit for the coarse-grained and fine-grained implementations as well. However, we faced challenges doing so for the fine-grained implementation. Hence, our fine-grained implementation uses an internal representation.

Approach

In order to analyze the performance of the lock-free BST implementation, we implemented three different approaches of BST: coarse-grained lock BST, fine-grained lock BST, and lock-free BST.

For all three implementations, we wrote the code in C++, with the use of <thread> library, <mutex> library. Specific APIs and functions used will be discussed in more detail below. Overall, we have a binary search tree parent class where all the child classes have three main operations: **search(), insert(),** and **delete().** Each node in the tree has an unsigned integer field with left and right child nodes. The machine that we targeted is the Intel based GHC machine that has hardware support for atomic compare-and-swap instructions. (Note: the BST is implemented as non-balanced tree)

External Representation of the tree

- For our lock-free implementation, we use the external representation of a bst. This greatly improves the performance of parallel operations since every node beside the leaf nodes is essentially for “routing” and the leaf node is where the data is stored. This comes with the tradeoff of using more memory of the number of nodes in external representation.

Coarse grained locking BST

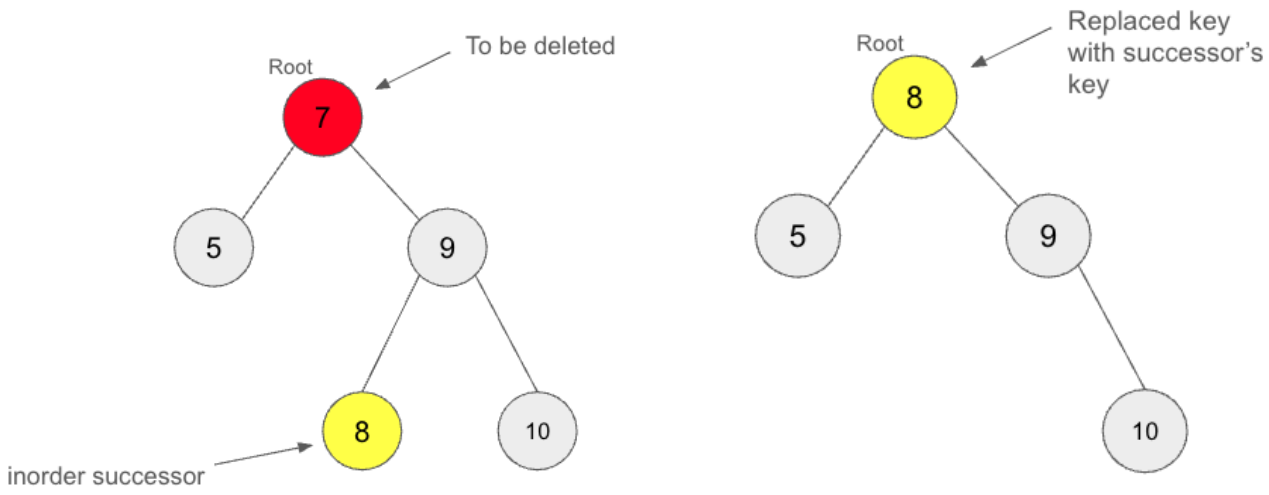
- For coarse grained BST, the idea is to protect the entire tree while a thread is performing any of the three operations. Therefore, a single global lock will be used. We used the **std::mutex** lock from the <mutex> library. This is a simple mutex lock that will make any other threads to be blocked and wait until the mutex is unlocked. This implementation is the most straightforward out of the three where the single global block will be acquired before an operation is allowed. The thread will release the lock after performing an operation.
- In order to minimize any other factors during analysis, we implemented the coarse grained bst as an external representation. This still follows the above idea where a global lock is used to lock the entire tree for operation. The only difference is that actual data is stored in the leaf nodes.

Fine grained locking BST

- For fine grained BST, we utilize the concepts taught in class from the lock-free lecture. We implemented the “hand-over-hand” locking approach from class. The goal is to reduce contention of a single global lock and allow other threads to perform operations on a different part of a tree that another thread is not working on. For the data structure used, we slightly modified the node structure to add a perNodeLock that also uses the

std::mutex. In addition to the perNodeLock, the single global lock is still needed. This global lock is for protecting concurrent root node accesses. This ensures thread safety when root node is null, root node is being deleted, and ensuring root is not modified while acquiring the next node's lock.

- For all three operations, it will always start by acquiring the global lock and checking the root. During the **search()** operation, one key idea is that before releasing the global lock or any locks, the code will always acquire the next node's lock before releasing the current lock. This also applies to other operations except for some corner cases. The **insert()** operation follows a similar structure to the **search()** operation. Additionally, the thread will need to ensure that the parent node's lock needs to be held while a new child is being created.
- For the **delete()** operation, instead of holding only the current node's lock, the current node's parent lock is also required to be held. The operation can be broken down into three general cases:
    - Deleting a node with no child node
        - This is a simpler case where the thread holds the lock of both the node being deleted and its parent node while it deletes the node and updates the parent's left or right pointer.
    - Deleting a node with one child node
        - This case is similar to the no child node case where the thread holds on both current and parent node's lock. The only difference is that the parent's pointer will be pointing to the single child node that current was pointing to.
    - Deleting a node with both right and left child node
        - This case requires finding the "inorder successor" of a node. This simply means the next smallest node that is greater than the current node. For example, if the current tree has data from 0~20 and the current node being deleted has a value of 10, the inorder successor node will have the value of 11. In code, this can be done by going right once and keep traversing to the left until leaf node is reached. The goal is to replace the current node with the inorder successor node. The challenge is essentially maintaining the locks of current node, inorder successor node, and inorder successor parent node. In addition, to prevent complex pointer operation, we replace the current node's key with the inorder successor node's key instead of deleting the current node while redirecting the inorder successor node's pointer. The figure below illustrates the idea described for deleting a node with both right and left child nodes.



Before Deletion (deleting 7) After deletion

Figure 1: Fine grained deletion of a node with two children

- Note: For fine grained locking, we did not implement the external representation. We try to keep the fine grained locking as external representation. However, we encounter several issues that prevent us from progressing with the project. Therefore, we implemented the usual internal representation. However, we did keep the attempts of implementation in the code as comments.

Lock free BST

- For the lock free version of the bst, we implemented from the references paper [1] with the provided pseudocode as starting point. As discussed in the above section, we implemented the lock-free version with external representation. The lock-free implementation also consists of the same three operations: **search(), insert(), and delete().** In order to preserve correctness of the bst without the use of lock during concurrent operation on the bst, special function is needed to achieve this.
- Two atomic operations that we used in C++
    - **__sync_bool_compare_and_swap**
        - GCC built-in function
        - This function is the atomic compare-and-swap (CAS) discussed in class. It will compare the value with an expected value. If the values match, it will replace the value with a new value. (return true if swap was performed, false if not).
        - Note: This function provides a full memory barrier to ensure no memory operations are reordered around the atomic operation.
    - **__sync_fetch_and_or**

        - GCC built-in function
        - This function performs an atomic fetch-and-or on a value. In particular, we use this function to perform the atomic bit test and set similar to the test and set discussed in class. This function will perform a bitwise or between the current value and the given value then stores the result back to the current value. The function returns the original value before the bitwise or operation.
- Before further discussing the lock-free algorithm, some additional notes are discussed below including the two helper functions. These information and the two functions are also discussed in the research paper [1].
    - This algorithm operates at the “edge level”. For example, if there are two nodes A and B, and we refer to the edge A->B as E. When both A and B are going to be removed from the bst, the edge (E) will be **“flagged.”** When only B is going to be removed from the bst, the edge (E) will be **“tagged”.** When the edge is flagged or tagged, it cannot be modified. This prevents concurrent modification by different threads.
    - **seek()**
        - The seek function essentially is very similar to the search function. It will traverse down the tree until the leaf node is reached. The traversed path from the root to the leaf is referred to as “access path” [1]. In addition, it stores four nodes during the traversal, which we referred to as seek record. The seek record contains four nodes: terminal, parent, ancestor, and successor. The terminal node is the leaf node and parent node is the termina’s parent node. For the ancestor node and successor node, there will be two cases. 1) Without concurrent deletions, the successor node **is** the parent node and the ancestor node is the successor node’s parent. 2) Nodes in the access path from successor node to parent node are being removed from the tree [1]. In order to guarantee the above concepts, the tree always started with three nodes/keys. For our code, we used UINT32_MAX (root), UINT32_MAX-2(root->left), and UINT32_MAX-1(root->right) as the three starting nodes.
    - The following graph is an example that we get from the research paper [1] to show an example of the four nodes with flagged and tagged edges.

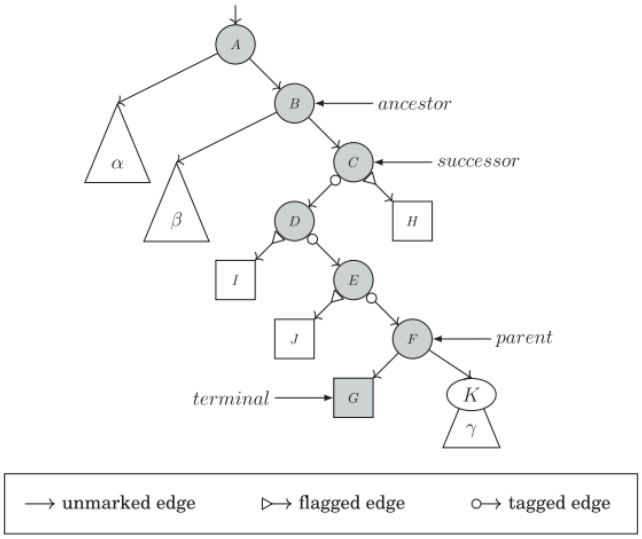
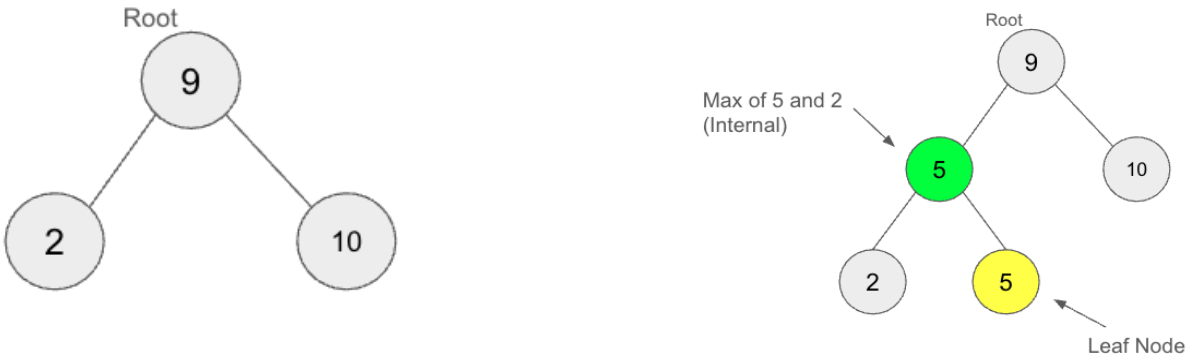


Figure 2: Example illustration from the paper [1]

- **cleanup()**
  - The cleanup function is similar to the normal delete operation in other bst. The cleanup function will take the seek record as the argument and delete both the leaf node and its parent from the bst. In this function, we used __sync_fetch_and_or for flagging and __sync_bool_compare_and_swap for modifying the memory address.

- With the above background information, we will discuss how the three operations utilizes the concepts.
  - **search()**
    - The search function is the simplest out of the three. A call to the seek function will return the seek record. The search function will check whether the terminal node's value matches the value that it is looking for
  - **insert()**
    - For the insertion operation, it also starts with calling the seek function to obtain the seek record. Since this implementation uses external representation, when a node is being inserted, two nodes need to be created: the actual data\leaf node and a new internal node that has a value equal to the maximum of the new value being inserted and the terminal node returned from the seek record [1]. An illustration is given below to illustrate the concepts.



Before inserting 5 **After inserting 5**

Figure 3: Lock-free insertion of 5

- __sync_bool_compare_and_swap is used in this function to modify the edge. If the compare and swap failed, the function restarted from the very beginning of obtaining the new seek record.

- **delete()**
  - For the deletion operation, it consists of two different modes. One mode (referred to as **injection mode** in the paper [1]) is responsible for "marking the edges" where the other mode (referred to as **cleanup mode** in the paper [1]) is responsible for "actually removing the **leaf node** and its **parent**" that is marked in the first mode [1].
  - Similar to the other operation, the delete also calls seek function to obtain the seek record. For the injection mode, it essentially marks the edge to the leaf node that is being deleted by "flagging". This is done using the __sync_bool_compare_and_swap function. If the compare and swap is successful, delete operation is guaranteed to be completed [1]. After flagging, it will transition to cleanup mode. Similar to the insert() operation, if the compare and swap failed, the function restarted from the beginning of obtaining the new seek record.

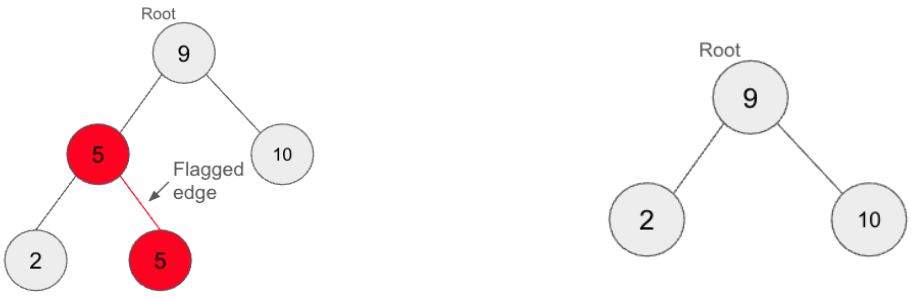


Figure 4: Lock-free deletion of 5

Testing for correctness / mapping to thread

- In order to test the correctness of the program, we wrote a correctness test that is able to test all three implementations. The correctness test function is called InsertSearchDeleteSearch.
    - Each thread will run a sequence of operations on a distinct range of keys but all threads will perform operation on the same tree. The key range for each thread is determined by its thread_idx and NUM_KEYS. The thread_idx * NUM_KEYS is the starting key for each thread..
    - The function implements a four stages test for all the keys in the key range.
        - Insert a new key into the BST
        - Search the newly inserted key (should found the key in the tree)
        - Delete the key from the BST
        - Search the key again (key should not exist in the tree)
    - The test uses assert statements during the two search stages to check for correctness.
- For creating and spawning threads, we use the C++'s thread library (std::thread). Each thread is created using std::thread and can be identified with the thread_idx. Each thread will call the same InsertSearchDeleteSearch function described above. The BST data structure will be passed by reference (std::ref) to allow all threads to operate on the same tree in order to check the correctness of concurrent operations on the same tree. Thread to core mapping is not defined explicitly. We allow the operating system to map threads to available cores.
- Besides the correctness test, we also implement other functions specifically to analyze the performance of all three implementations which will be discussed below in the results section.

Results

Depending on the application and its requirements, BSTs vary in size. Hence we decided to run our experiments for various key space sizes. For the various key space sizes, the value of keys used is [0, KEY_SPACE_SIZE). We considered a uniform key distribution in which all keys in the key space have the same probability of occurrence. We achieved this using std::mt19937, which is a Mersenne Twister pseudo-random generator, and std::uniform_int_distribution, which produces random integer values uniformly distributed on the specified closed interval. The maximum degree of concurrency or contention depends on the number of threads that can concurrently operate on the tree. We varied the number of threads from one to twice the maximum number of hardware threads supported by the GHC machine in suitable increments. We considered three different types of distributions:

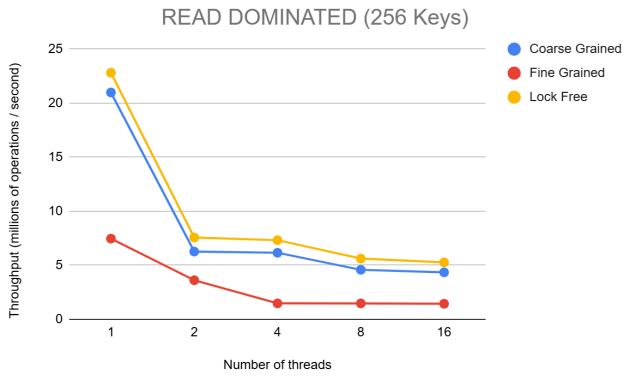
1) Read-dominated: It contains 90% search, 5% insert and 5% delete operations.
2) Balanced: It contains equal fractions of search, insert and delete operations.
3) Write-dominated: It contains 0% search, 50% insert and 50% delete operations.

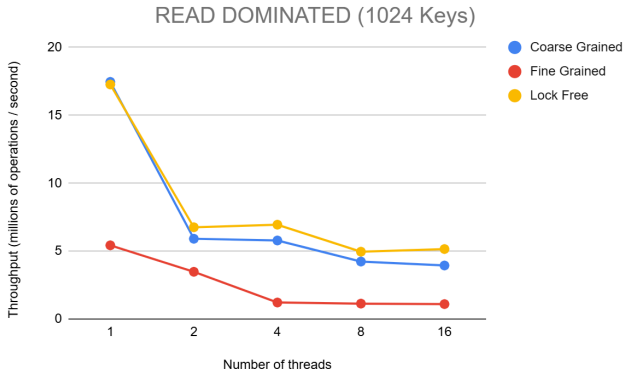
We created a single threaded test to verify correctness. This test involved generating 1000000 keys in the range [0,100000). This increased the changes of key values being observed multiple

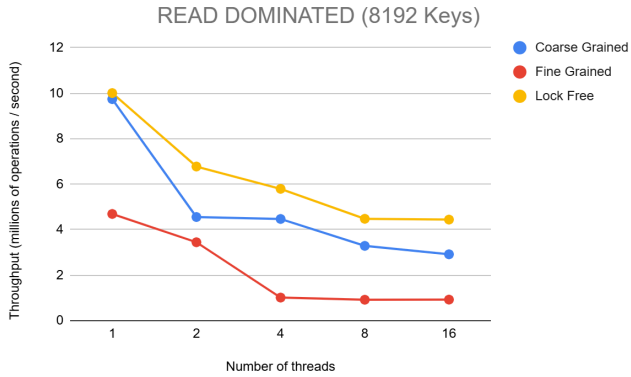
times and different operations being conducted on them. We used a separate boolean vector to keep track of if a key was inserted(true) or removed(false). Once the operations on the keys were complete, the result of a search of each key was compared against its entry in the boolean vector. By recreating specific sequences of operations on specific keys and visualizing the tree, we were able to identify bugs and fix issues in our implementations.

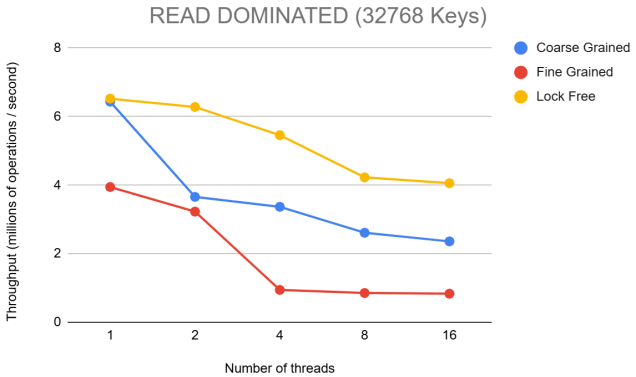
We created multi-threaded tests to verify both correctness and measure performance. Multi-threaded correctness was tested by having threads insert, search, delete & search for keys in non-overlapping ranges. Using non-overlapping ranges allowed each thread to check using search operation that a key inserted by it should be present in the tree and a key deleted by it should not be present in the tree.

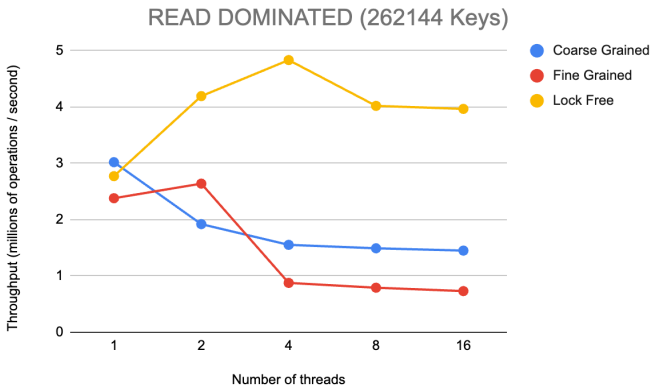
We used throughput as our measure of performance. Throughput is measured in millions of operations per second. We kept track of the number of operations executed by atomically incrementing a counter for each operation executed. Workloads were run for 10s. We calculated an average of 5 runs for each workload. To capture only steady state behavior, we pre-populated the tree to 50% of its maximum size prior to starting a run with random keys. Below, we plotted graphs for each workload for various key space sizes. Each graph shows the throughput of each implementation for various thread counts.

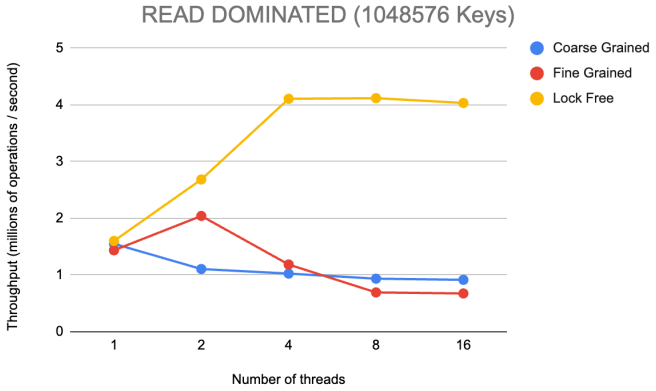




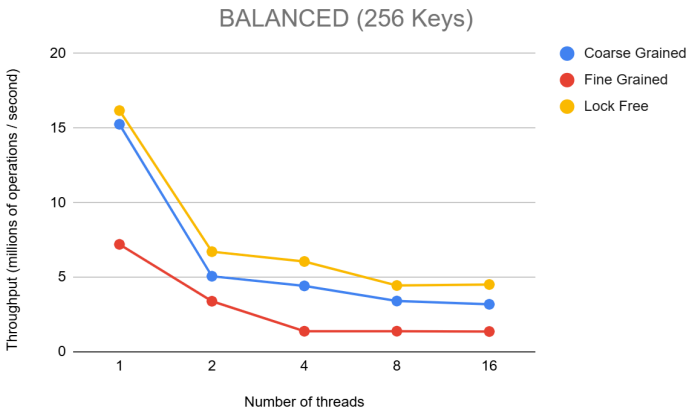


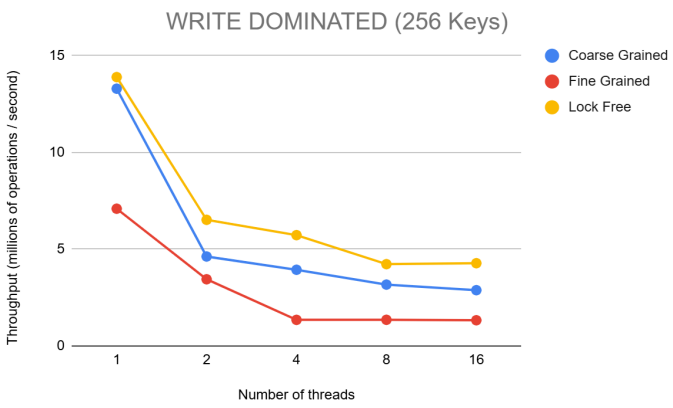






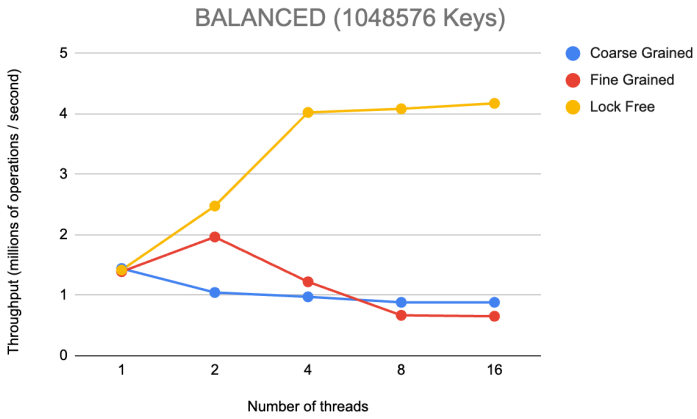
For read-dominated workload, lock-free performs better than other implementations for all key space sizes. The improvement in throughput from both lock-based implementations increases as key space size increases. This is probably because a bigger key space size allows for a greater degree of concurrency. More threads can simultaneously make forward progress if they are operating on non-overlapping areas of the tree. The global lock for the root in both lock-based implementations is the bottleneck in exploiting concurrency. For key space sizes of 256, 1024, 8192 & 32768, the throughput of lock-free drops as the number of threads increases. This might be due to the dependencies on parent and child nodes which causes the working set of threads to overlap, preventing true parallelism and stalling the threads that arrive later. With bigger key space sizes of 262144 & 1048576, we generally observe that the throughput increases as the thread count increases. The throughput stagnates from 8 to 16 threads because the GHC machine has 8 cores and the time gained from having 16 threads is lost due to the overhead of context switching.

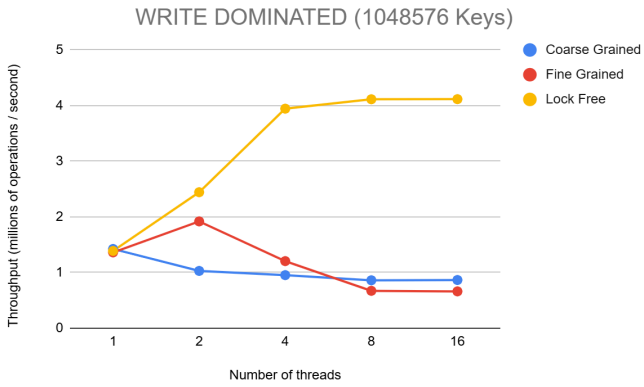




Trends for balanced and write-dominated workloads for all key space sizes are similar to that observed for read-dominated workload. Read-dominated workload has higher throughput than balanced and write workloads because it consists mostly of lookups that do not require

tagging/flagging nodes, in the case of lock-free implementation, or hold locks for much shorter duration, in the case of coarse-grained and fine-grained implementation.

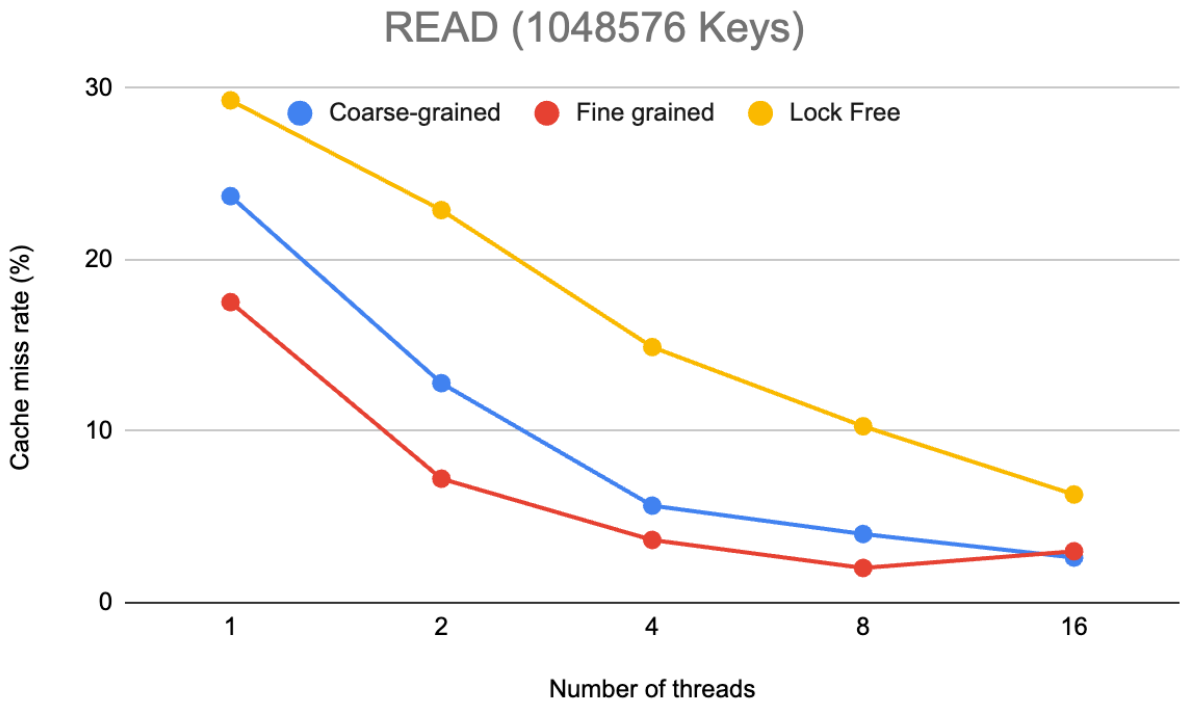




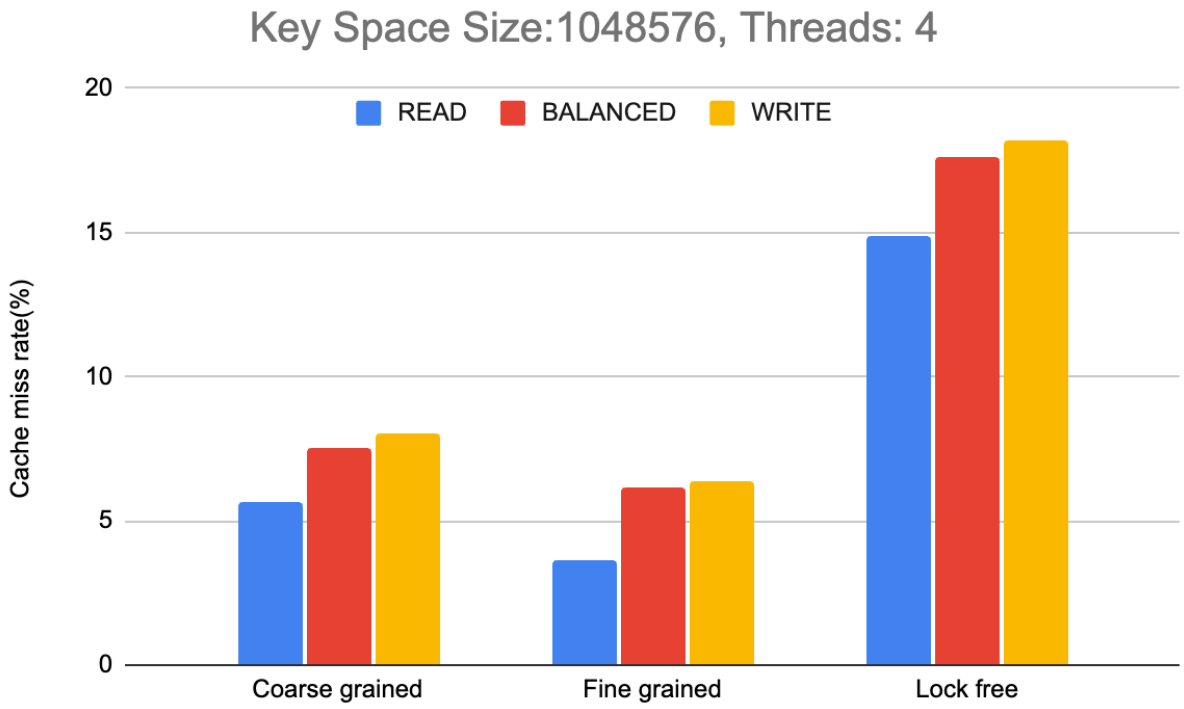
At higher key space sizes, all implementations have a very similar trend for throughput regardless of the type of workload. The type of workload seems to have a greater impact on throughput at smaller key space sizes as there is a greater difference in throughput across the different workloads for a specific implementation with a specific thread count. This can be attributed to minimized contention between threads due to the massive key space size and vice versa. We observed that there is marginal change in throughput from 4 to 8 threads. We speculate that there would be a noticeable increase in throughput from 4 to 8 threads if the tree size were even bigger.

An increase in throughput implies an increase in speedup. Speedup increases with higher thread counts at larger key space sizes. Despite the implementation complexity, the benefits of a lock-free BST can be realised for applications that have huge key space sizes and utilize multi-threading. A simple coarse-grained BST would work just as well as a lock-free BST for single-threaded applications for all key space sizes.

We also investigated the cache performance of our implementations by observing the cache miss rate. To obtain the most accurate results, we initially wanted to measure the cache miss rate for the duration that the threads were running. We tried to do this by using `perf_event_open` to enable/disable monitoring only around the part of code where threads were running. However, we found the tool very complicated to use and we were unsure if the results that we were observing were that of the main process only or also included the threads. In the interest of time, we use perf stat to capture the cache miss rate and cache references of the entire program. This time, we did not pre-populate the tree to take the initial cold misses into account and ran the test for 20s to more stable readings.



We see from the plot that between coarse grained, fine grained, and lock free implementations, the fine grained has the lowest cache miss rate. This is what we expected. As discussed in the approach section, our coarse grained and lock free implementations use external representation of BST while fine grained implementation uses internal representation. Internal representation will have a lower cache miss rate because it has less number of nodes than external representation. There will be no “routing nodes” and data can be stored anywhere in the tree, not just the leaf nodes. We also see that the miss rate decreases as the number of threads increases. This is due an increase in data sharing between threads, i.e. nodes near the root of the tree are frequently accessed and with more threads, these nodes are even more frequently hit in cache. We also observe that lock-free has a higher cache miss rate than coarse-grained. This could be attributed to the fact that there could be many calls to __sync_compare_and_swap when threads try to insert or delete, which causes more invalidations than trying to grab a lock.



We also take an example of running on 4 threads. We see that from the plot write dominated always has the highest cache miss rate and read dominated has the lowest cache miss rate, regardless of the implementation. This is also expected since write dominated tasks have a lot more insert and delete operations. Insert and delete operations do extra modifications on top of basically doing a search operation. This can lead to more evictions and invalidations by other cores, thus leading to higher cache miss rate.

References

[1] Aravind Natarajan, Arunmoezhi Ramachandran, and Neeraj Mittal. 2020. FEAST: A Lightweight Lock-free Concurrent Binary Search Tree. ACM Trans. Parallel Comput. 7, 2, Article 10 (June 2020), 64 pages. https://doi.org/10.1145/3391438

Distribution of Credit

Apurva: Coarse-grained, lock-free search/insert/cleanup, tests, report
Harvey: Fine-grained, lock-free delete, tests, report
Credit distribution: 50%-50%