Lock-free Binary Search Tree and Skip List
Project Proposal

Name: Chenhao Li; Andrew ID: chenhao3

March 22, 2022

1 Summary

I will complete this project alone. I believe the workload is suitable for me.

I will implement a lock-free binary search tree and skip list, and will test their correctness and evaluate their performance under different workload patterns and configurations on multi-core CPU platforms.

2 Background

Dictionary is an important abstract data type in real-world applications. The most basic Dictionary needs to support `search`, `insert` and `remove` operations for given keys. It can also be extended to associate value(s) for a specific key.

Binary search tree (BST) and skip list are two efficient implementations for Dictionary when a partial order relationship is defined on the key set. BST associates each key with a node, and organizes the nodes in a tree structure. In the original BST, the key in the parent node should be strictly greater than keys in the left subtree, and strictly smaller than keys in the right subtree. This allows Dictionary operations to only explore one branch based on the comparison between the key in the current node and the given key.

The asymptotic time complexity of tree operations is proportional to tree height. If the tree is properly balanced, it will be $O(\log(n))$, where n is the number of tree nodes. However, the tree can degenerate into a long chain, in which situation the operations will take $O(n)$ time. This situation is easy to achieve, such as continuously inserting incremental keys into the tree. To solve this problem, there are many techniques to rebalance the tree, such as Red-black Tree and AVL Tree. They usually require the rotation operation involving multiple nodes.

Skip list, on the other hand, is a linear structure. It consists of multiple layers of ordered singly linked lists. The number of layers in a node is determined by an exponential distribution based on a certain probability. Each node stores a set of `next` pointers equal to its number of layers. In this way, the bottom layer is an ordinary linked list, and higher layers are connected to skip multiple nodes.

Such a probabilistic data structure allows $O(\log(n))$ operations on average. It is possible, in the worst case, that the operations also take $O(n)$ time. However, this situation is very rare, and it is not because of any specific operation pattern, but because of the bad result of the random number generator (really bad luck instead of malicious user input).

Many papers and applications suggest that skip list is easier to parallelize than BST because it doesn't involve non-local rebalance operations. This is the reason why I chose to implement these two data structures. I hope to compare their implementation difficulty, running speed and memory overhead, so as to analyze whether skip list is really a better choice of a concurrent dictionary.

3 Challenge

The main challenge is to correctly parallelize the modifications (`insert` and/or `remove`) without locks. Binary search tree needs to maintain the pointer links between ancestors and children very carefully. Skip list is expected to be relatively simpler, but also involves maintaining a synchronized state among multiple layers. They will mostly be implemented via compare-and-swap (CAS) instructions, which can be very tricky to reason about and debug with.

Another concerning challenge is the ABA problem and garbage collection. I will use C++ for my implementation, where memory allocation and deallocation are manual. Since the data structures are lock-free and multiple operations may be in progress at the same time, CAS instructions may be confused if a pointer is freed by one operation and reused by the system memory allocator in another. I plan to design a garbage collection module to solve this problem.

There are also other concerns. For example, spatial locality and false sharing may both affect the performance, and it is hard to optimize for both of them. It is also challenging to design "typical" workloads that are able comprehensively to test the correctness and measure the performance.

4 Resources

I will use no starter code and build the project from scratch. Currently I decide to build the lock-free binary search tree based on [NM14], and build the lock-free skip list based on [Fra04]. But I may also switch to other papers with more in-depth literature reviews. I will test my implementations on GHC and PSC Bridges-2 Regular Memory machines.

5 Goals and Deliverables

- Implement a baseline concurrent dictionary with STL data structures protected by a readers-writer lock.
- Implement a lock-free binary search tree, with `search`, `insert` and `remove` operations. According to my experience and literature reviews, the complexities of these three operations are likely to increase in order.
- Implement a lock-free skip list, with `search`, `insert` and `remove` operations.
- Implement a garbage collection module to enable the previous two data structures to reclaim memory.
- Measure the performance of the previous three data structures under different workload patterns and configurations. The garbage collection module can also help measure memory consumption. These measurements are likely to produce plots suitable for my poster session. Hopefully, I will be able to discover come up with some observations and insights in the trade-off in these two data structures through experiments.

The 75% goal is everything except the garbage collection module. In this case, the data structures are likely to still function correctly with a similar performance, but they will cause memory leaks. The 125% goal includes implementing binary search tree and skip list with fine-grained locks and taking them into the measurement. They may not be technically more difficult, but not my mainline goal.

6 Platform Choice

Concurrent dictionaries are most often used in multi-core CPU machines. The examples may include web servers, database management systems, or even operating system kernels. It is a common

practice in these applications running on multi-core CPU machines to concurrently query and update keys in a dictionary. Another reason is that implementing these data structures requires the use of complex atomic instructions to manipulate the global memory, which is usually only possible or much more effective on modern CPUs.

7 Schedule

- Week 1 (3.21 - 4.03): Do more literature reviews and determine the implementation logistics of the lock-free binary search tree and skip list. Implement the baseline concurrent dictionary with STL data structures protected by a readers-writer lock.
- Week 2 (4.04 - 4.10): Implement the lock-free binary search tree without memory reclamation.
- Week 3 (4.11 - 4.17): Design correctness test suites. Implement the lock-free skip list without memory reclamation.
- Week 4 (4.18 - 4.24): Implement the garbage collection module and memory reclamation in the previous two data structures.
- Week 5 (4.25 - 5.01): Design performance test suites and collect measurement results. Finish up the project, write the final report and poster pages.

References

[Fra04] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 317–328, New York, NY, USA, 2014. Association for Computing Machinery.