Implementation of a Concurrent Hash Map using Split-Ordered Lists

Brett Bissey
Department of Computer Science
University of Central Florida
Orlando, Florida, USA
brettbissey@knights.ucf.edu

Christopher Feltner
Department of Computer Science
University of Central Florida
Orlando, Florida, USA
chris.feltner@knights.ucf.edu

George Lu
Department of Computer Science
University of Central Florida
Orlando, Florida, USA
georgelu@knights.ucf.edu

Jesse Rehrer
Department of Computer Science
University of Central Florida
Orlando, Florida, USA
jesserehrer@knights.ucf.edu

Sarah Wilson
Department of Computer Science
University of Central Florida
Orlando, Florida, USA
sarahwilson@knights.ucf.edu

Abstract

Hash tables are a fundamental data structure for storing key-value pairs which provide constant time contains(), insert(), and delete() functionality. In a multi-threaded environment, there is the additional consideration of maintaining consistency and correctness among threads, while also avoiding contention. While one way to maintain consistency is to use locks, this can lead to contention among threads for the shared resource and reduce performance. In this paper we implement a lock-free concurrent hash map using a Split-Ordered List, which maintains correctness in a multi-threaded environment while also not requiring threads to wait for other threads to access the resource.

I Introduction

Hash tables are data structures that consist of key-value pairs. The methods for a sequential hash table are typically insert(), delete(), update(), read() in O(1) time. The main reason why hash tables have this fast performance is because they typically have an array-like structure and items are indexed by a hash value of their keys. The hash table can directly access its keys which allows most of its methods to have a constant runtime. Hash tables typically have comparatively worse performance when it has to deal with collisions of duplicate hashed values, or when it has to expand. Resizing of a hash table usually consists of re-mapping all of the hashed key value pairs which is an expensive operation.

The performance of a sequential hash table can be improved by making the data structure concurrent. Since hash tables are one of the most common data structures, it is important to take advantage of an environment that has multiple cores. Creating a hash table which can perform well in a concurrent environment can improve the performance of multithreaded software which uses it by reducing the amount of code that must be executed sequentially. The basic foundation for our implementation of a hash map is a concurrent list-based set. This list is combined with a set of pointers that can skip to different sections of the list. These pointers make searching for and accessing the different keys in the hash table more performant. Since most of the operations rely on these two actions, the runtime of all methods is based on an efficient operation.

The two most challenging parts of parallelizing this data structure are ensuring the hash table remains correct while all the threads are working on it, and minimizing the amount of waiting the threads have to do in order to ensure this correctness. Our implementation is lock-free. Having no locks allows the threads to work on the hash table concurrently. In order to ensure correctness, we rely on compare and swap operations which allow us to check the expected value before we change it.

II Background

The following are terms used throughout the paper [2]:

Wait-free: ”The guarantee that every call finishes execution in a finite number of steps [2].”

Lock-free: ”Guarantees that infinitely often some method call finishes in a finite number of steps [2].”

Linearizable: A set of operations’ ordered list of invocations and responses can be extended by adding responses such that the resulting extended list can be re-expressed as a sequential history and the sequential history is a subset of the original list.

Compare and Set/Swap operation: An atomic instruction that compares the current value of a variable to the value that it is expected to be. If the expected value matches the current value, then the value is updated to the desired change.

Non-blocking Synchronization: This approach eliminates locks, and it instead uses atomic operations like Compare and Set (see above). If there is a thread actively working on the concurrent object, it is guaranteed that the same or another thread will complete in a finite number of steps, no matter the operations of the other threads.

Lazy Synchronization: This approach postpones operations which could delay threads. An example of this would be a logical deletion of an element before physically deleting the element.

Correctness: For a concurrent object to be correct, concurrent access to that object must be synchronized.

III Related Works

A. Split Ordered Lists

Shalev and Shavit [6] made a lock-free and wait-free hash table based on Michael's work in the section below. The hash table was made using a concurrent list data structure. The list had sub-lists (or buckets) that are organized by the MSB bits of the keys. A separate array keeps the pointers to dummy nodes demarcating the buckets. This way, using the hash value of the key, the algorithm can use the index of the bucket reference array as a "shortcut" into that sub-list in the larger linked list, making the access time O(1). The authors used recursive split ordering to organize the nodes. This method only requires one compare and swap which keeps the runtime constant. To expand the hash table, new references are created to generate a new sub-list, but importantly, the order of the items in the list does not change.

The core idea of the Split-Ordered list is to move the "buckets among the items" rather than the "items among the buckets [6]." By only editing the locations of the references on list expansion, it gives better performance than having to move the individual items in each sub-list between references. A problem that we hope to investigate and possibly improve is the lack of a method to reduce the number of buckets when the list contracts if sufficient items are deleted. Currently, all of the allocated bucket references and dummy nodes remain. This means that the Shalev & Shavit implementation could use more memory than it actually needs.

B. Lock-Free hash tables and List Based Sets

Michael [5] created a lock free list-based set that was used as the foundation for the Shalev & Shavit implementation of Split-Ordered Lists. This paper improved on the previous parallel programming research that used locks. Considerable overhead was needed to reorganize those lock-based lists because all of the locks needed to be held at the same time, preventing other operations from doing work on the list. This list data structure primarily uses Compare and Swap operations to achieve synchronization between the threads. All of the operations on the list are linearizable, meaning that the list operations can be used in another data structure's operations while maintaining linearizability. Nodes that have been deleted are first logically deleted by marking a node. If the algorithm finds a marked node during its traversal, it will then remove it from the list, and prepares the deleted node to be reused in the list. This paper also documents the first lock-free hash table that is built on top of the lock-free list based set.

C. Dynamic-Sized Non-blocking Hash Tables

Liu, Zhang, and Spear [4] made a Dynamic-Sized Non-blocking Hash Tables. They point out an issue in the Shalev and Shavit's split ordered list: When Shalev and Shavit implementation expands, there are dummy nodes that are permanently inserted into the linked list. The problem is that when the hash table needs to shrink, it is unclear how the marker nodes are reclaimed.

In addition, they point out that the Shalev and Shavit implementation assumes memory size is bounded and known and relies on tree based indexing. This dynamic sized non-blocking hash table introduces a new re-sizable hash table implementation that fixes the resizing problem. Compared to the split ordered list, the dynamic non blocking hash table will have 3 new properties. It will be dynamic so the bucket array can adjust it's size. The bucket array is unbounded so no assumptions is made about the size of memory. The hash table will be wait-free for every operation of insert, remove, and contain. In the experimental evaluation they showed that the hash table achieved low latency and high scalability; desirable characteristics of a hash table.

D. Almost Wait-free Resizable Hashtables

Gao, Groote, and Hesselink [1] created a hash table that is almost wait free. The paper discussed how important lock and wait free algorithms are since locks and mutexes create added overhead on the runtime. This list is not wait free because when the list needs to expand, the threads all need to wait for a new hash table to be created and initialized from the old hash table. This algorithm can scale, and it scales up linearly depending on the number of processes. This paper unfortunately did not contain data that demonstrates the performance of their algorithm.

E. Wait Free Hash Map

Laborde, Feldman, and Dechev [3] made a wait free hash map. The 7x improvement over a blocking design is attained through compare and swap operations that maintain the correctness of the hash map, while also providing better performance than locks would. This hash map has a tree-like structure because of the sub-arrays used within the hash map. These sub-arrays reduce the need to wait, and it also uses a perfect hash function to give each element a unique position. The data structure expands whenever a node is contended to a certain length that is a predetermined power of two. This predetermined size is used for all node expansions. The data structure also may be expanded if there is a lot of contention in a certain area of the hash map. There is also a global watch array that a thread can check before they perform an operation on a node so that two threads do not operate on the same node at the same time. This hash map has wait freedom because their methods all return in a bounded number of steps.

IV The Algorithms

A. Lock Free Linked List Based Set

Split-ordered lists use a lock-free linked list structure to store values. Following the example of Shalev & Shavit [6], we implemented Michael's [5] design in Java. This list uses compare and set operations instead of locks in order to ensure that the data structure is correct. The algorithm uses lazy synchronization, marking nodes as logically deleted before they are physically removed from the list. Michael's design uses marked pointers, but in Java it is not possible to directly change the value of references. We resolve this by using the AtomicMarkableReference provided in java.util.concurrent for the references linking nodes in the set. We can depend on the Java Virtual Machine to run garbage collection on nodes deleted from the list. Java garbage collection is concurrent, and will run while the algorithm is running, so the memory of deleted nodes will be de-allocated at the time most convenient for the Java Virtual Machine.

1) Node Class

We create a class to represent a node in the list. In Michael's implementation, each node in the list has a markable pointer to enable lazy synchronization for the delete operation. Our Node has an AtomicMarkableReference<Node> pointing to the next node in the list. AtomicMarkableReference has a reference to an object, along with a boolean value for the mark, which can be changed atomically. The key is currently implemented as an int, but will be changed to a bitset to accommodate the key values of a Split-Ordered list. Finally, Node has a boolean value to mark if it is a dummy node, which is necessary for Split-Ordered list.

2) Find

The find method traverses the list trying to find the desired key. This method is used in the insert method to check if the node is already in the list, to disallow repeat elements. Find returns false if the key is not there, and returns true if the key is found in the list. While traversing, if find()'s previous node has been marked for deletion, it will restart to ensure that the node is still accessible from head. When find comes to the target key, it returns true. If find encounters a key that is bigger than the target or it encounters the end of the list, it returns false. Currently, find() also tries to physically delete any logically deleted nodes. This means that our list is frequently removing logically deleted nodes, which slows down find().

3) Insert

The insert operation first calls find() on the key given to insert to determine if the element is already in the list. If the same key is found then the list will return false. The lock-free set will then create a node with the given key, initializing the marked reference to false. Then the lock-free set will insert the node at the head of the list. We also provide a function insertAt which takes as parameters a node to insert as well as a reference to a node to insert after. This separates the logic of traversing to find the insertion point, which should be taken care of by the Split-Ordered List, from the lock-free set.

4) Delete

The delete operation begins by performing the same logic as find, without the physical deletion. We traverse the list beginning at the head until the current node's value is greater than or equal to the value we are trying to delete. If it is greater than the value we are trying to delete we return false, since the node is not in the set. If the list has been altered in any way that affects this traversal the deletion restarts. The delete operation utilizes Java's AtomicMarkableReference in order to perform a two step deletion. First logical deletion is performed by using CompareAndSet to set the mark bit to true. This indicates that the node has been logically deleted from the list. We then try once to physically delete the node from the list by performing CompareAndSet to set the previous nodes next field to the current nodes next field. If the CompareAndSet operation fails, the method just returns true since the find method will repeatedly attempt physical deletion of a node until it succeeds. We also plan to provide a function deleteAt which takes as parameters a key to delete as well as a reference to its previous node. This separates the logic of traversing to find the deletion point in the Split-Ordered list from the lock-free set implementation.

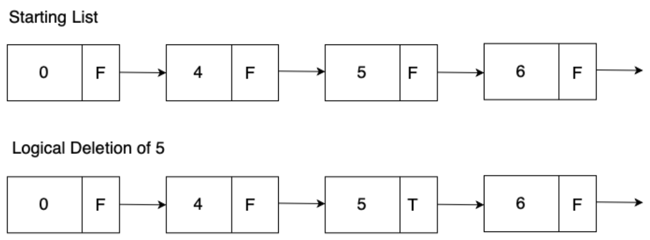

Fig. 1. Logical deletion of a node

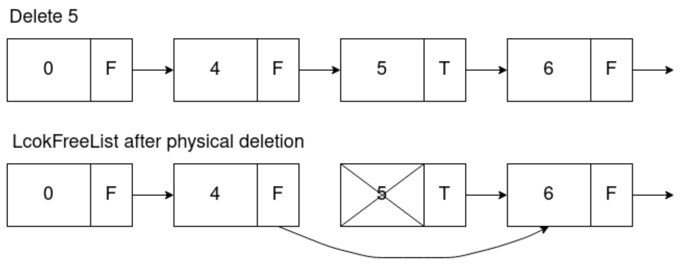

Fig. 2. Physical deletion of a node from the Lock-Free Set

B. Concurrent Hash Map

We will implement a concurrent hash map similar to the Split-Ordered List with further optimizations from newer literature [6]. There is a Lock-Free List underlying the concurrent hash map's structure, as well as a dynamically adjusted array of buckets which holds references to different positions in the list, which is implemented using an ArrayList. The bucket that any particular node should be placed after depends on the modulo division of the key and the current size of the bucket ArrayList. If the bucket has never had any nodes in it, we must initialize the bucket and a dummy node to which the bucket will point. Shalev & Shavit mark dummy nodes by changing the least-significant bit of the node's reversed binary string key to 1. In our implementation, we store this dummy node status as a boolean value in the Node object.

1) Initialize Bucket

The initialization of buckets and the resizing of the ArrayList are necessary components in controlling distribution across buckets. Whenever we insert something that requires a bucket initialization, we create a dummy node that will serve as our modulo bucket reference and then insert afterwards. This Bucket initialization is almost always paired with dummy node creation; unless the dummy node is not yet physically deleted from the list and could be used again if the logical delete was undone.

2) Insert

3) Delete

4) Contains

V Results

VI Conclusions

References

[1] H Gao, J.F. Groote, and W.H. Hesselink. Almost wait-free resizable hashtables. In *18th International Parallel and Distributed Processing Symposium*, 2004.

[2] M. Herlihy and N. Shavit. The art of multiprocessor programming. In *Morgan Kaufmann*, 2008.

[3] Pierre Laborde, Stephen Feldman, and Damian Dechev. A wait-free hash map. In *International Journal of Parallel Programming*, 2017.

[4] Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized nonblocking hash tables. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 2014.

[5] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 2002.

[6] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 2006.

VII Appendix

A. Challenges

One of the challenges we faced was deciding on an implementation of concurrent hash tables. There are many research papers that have created concurrent hash tables, each one prioritizing a certain property, such as being wait-free, having low false positives, or using a certain hash function. This required us to evaluate the benefits and tradeoffs of different designs. Since most of the research we encountered used C-based pseudocode, it was a challenge to figure out the equivalent in Java. For example, we had to use AtomicMarkableReference to replace marked pointers that were used in the list data structure. We preferred Java over C/C++ because our team had more prior experience with Java.

Additionally, it was a challenge to improve the quality of the pseudocode used in the papers on which we base our implementation. For example, the Michael list-based set uses goto statements and functions which manipulate global variables, making their interactions with other functions unclear. We redesigned the algorithm to eliminate the use of goto and make program flow more clear.

Tasks	**Status**
Problem Definition and Team Declaration	Done
Literature Review	Done
Implement Lock-Free Set	Done
Build Split-Ordered List Operations	In Progress
Mid-Term Research Report	Done
Make Presentation	Not Started
Presentation	Not Started
Finish Split-Ordered List Operations	Not Started
Conduct Performance Experiments	Not Started
Visualize and Report Results	Not Started
Final Report	Not Started

TABLE I. Tasks for our project