An Analysis of Various Concurrent Hashing Algorithms

Jason Cai (jsc3234) and Tiffany Yang(tyy68)

Abstract—*In this paper, we present our implementation of concurrent versions of various hashing algorithms. We describe each of the hashing algorithms that provide the basis of our concurrent implementations and how they handle collisions. We also present a design alternative that was considered during our implementation. To evaluate our implementations, we perform common hashtable operations in parallel and measure the execution time for each and evaluate the results.*

Keywords—*hashing algorithms, concurrent algorithms, synchronization, hashtables*

I. Introduction

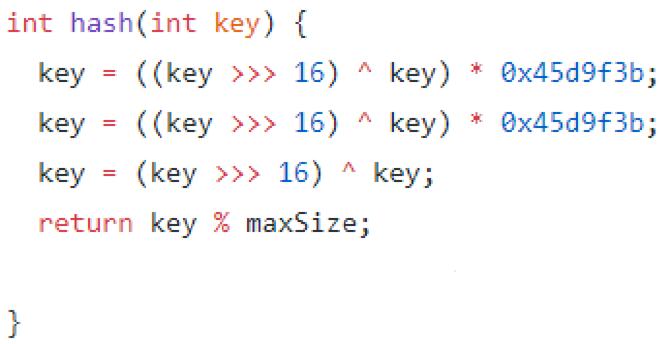
Key-value stores, which are usually implemented as hash tables, are a significant component of the workload for large-scale computing efforts. For technological companies to be able to provide adequate service to their users, concurrency is key. This study aims to investigate the differences between various concurrent hashing implementations as well as their performance overall.

The chief difference between various hashing algorithms concerns how hash collisions are dealt with. For naive implementations, collisions are dealt with by chaining together keys that map to the same hash. In the worst case, this behavior results in a linear search of the table. This paper will describe the various hashing algorithms that we use and their performance.

The rest of the paper is organized as follows. In section II, we provide the project description and how the hashing algorithms we used deal with collisions. Section III discusses some of the design alternatives we considered in our implementation and our reasoning as to why we didn’t choose that alternative as our final design. In section IV, we present the performance of our various concurrent hashing functions and evaluate those results.

II. Project Description

In this project we seek to implement Chain hashing, Cuckoo hashing, Hopscotch hashing, and Robin Hood hashing for use in a concurrent hashtable. We implemented each of these hashing methods using 3 different synchronization methods: coarse-grained locking, fine-grained locking, and lock-free. To establish a baseline for our implementations, we also made 2 base cases using the existing Java methods for concurrent hashtables, ConcurrentHashMap and Collections.synchronizedMap.



```
int hash(int key) {
  key = ((key >>> 16) ^ key) * 0x45d9f3b;
  key = ((key >>> 16) ^ key) * 0x45d9f3b;
  key = (key >>> 16) ^ key;
  return key % maxSize;

}
```

Figure 1: Hash function used in evaluation

For each implementation, we tested the performance of each hashing algorithm by timing how long each takes to add 3000 integers, get each integer, and remove 3000 integers.

For each of these operations we created 3 threads that would operate on 1000 integers each. In each implementation we set the hash for each integer to the hash function seen in figure 1. We chose this hash because it results in a fair amount of collisions within each hashtable, allowing each implementation to handle the collision in the way that is dictated by the hashing algorithm.

The rest of this section describes each of the hashing algorithms that we used in this study and how they handle collisions.

A. *Chain Hashing*

Chain Hashing has a relatively simple manner of dealing with collisions. Whenever two elements have the same key, the element to be added is appended to the previous element in the chain similar to a LinkedList. The more collisions that occur on a specific key, the longer the list at that location becomes and in the worst case becomes an array containing all the elements in the table.

B. *Cuckoo Hashing*

Cuckoo hashing is unique in that it maintains two hash tables with two different hashing functions. Each key maps to two values, so if an element is present in the table, it will be located at one of two locations. This behavior means that lookup and deletion can be done in constant time, and it is provided by amortizing the cost of probing values over insertion.

When element A is inserted into the table, its key is computed using the first hash function, and the element is placed in that location of table 1. If the location was empty initially, the function returns. If element B was already present in that location, element B is evicted, and the second hash function is used to calculate its location for table 2. This process repeats if there is another collision, and elements are shifted between table 1 and table 2 until no element is evicted. If a cycle occurs, new hash functions are computed, and the elements are re-inserted into the table using the new functions.

C. *Hopscotch Hashing*

Hopscotch hashing is inspired by cuckoo hashing but also has similarities with chaining and linear probing. The hash key of an element is used to calculate the bucket it belongs to. Hopscotch hashing maintains the notion of a neighborhood of fixed size H. If element A hashes to bucket i, if it is present in the table, it will be in a bucket j between i and i + H. Unlike a linear search with chaining, the number of buckets that will be checked to determine the presence of an element is capped at H. Value insertion works like Cuckoo hashing. Beginning from its bucket i, a linear search is performed until empty bucket j is found. If that bucket is within the neighborhood of i, A is inserted and the function returns. If not, a value B between i and j that belongs about j is swapped into j, and the swapping process continues for the empty bucket left by B.

D. *Robin Hood Hashing*

With each collision in a Robin Hood Hashing algorithm, a check is performed on the existing value to see how far it is from its preferred position. If the distance is greater than or equal to the allowed maximum, the existing value is left alone, and the next key is checked to place the new element. If the distance is less than the allowed maximum, the new value is placed at the key and the previous value becomes the value that needs to be inserted into the table. This process repeats until an insert occurs where there is no element that needs to be displaced.

E. *Java Concurrent HashMaps*

Java's hashmaps have a relatively unique way of dealing with collisions. When a collision occurs it is first treated like collisions are treated in the Chain Hashing Algorithm. However, starting from Java 8, once the collisions in a table for a specific key exceeded the class-defined constant, TREEIFY_THRESHOLD, the LinkedList at that key is converted into a balanced binarytree. This reduces this worst-case look- up time from O(n) to O(log n).

Java's ConcurrentHashMap and Collections.synchronizedMap are simply thread-safe versions of Java's hashmaps. The difference being the level of synchronization that each provides. For ConcurrentHashMap, only the current key that is being operated on is locked to a thread. Whereas, for Collections.synchronizedMap, the hashmap is treated like a key and a thread is required to own the hashmap before it can perform any operations on it.up time from O(n) to O(log n).

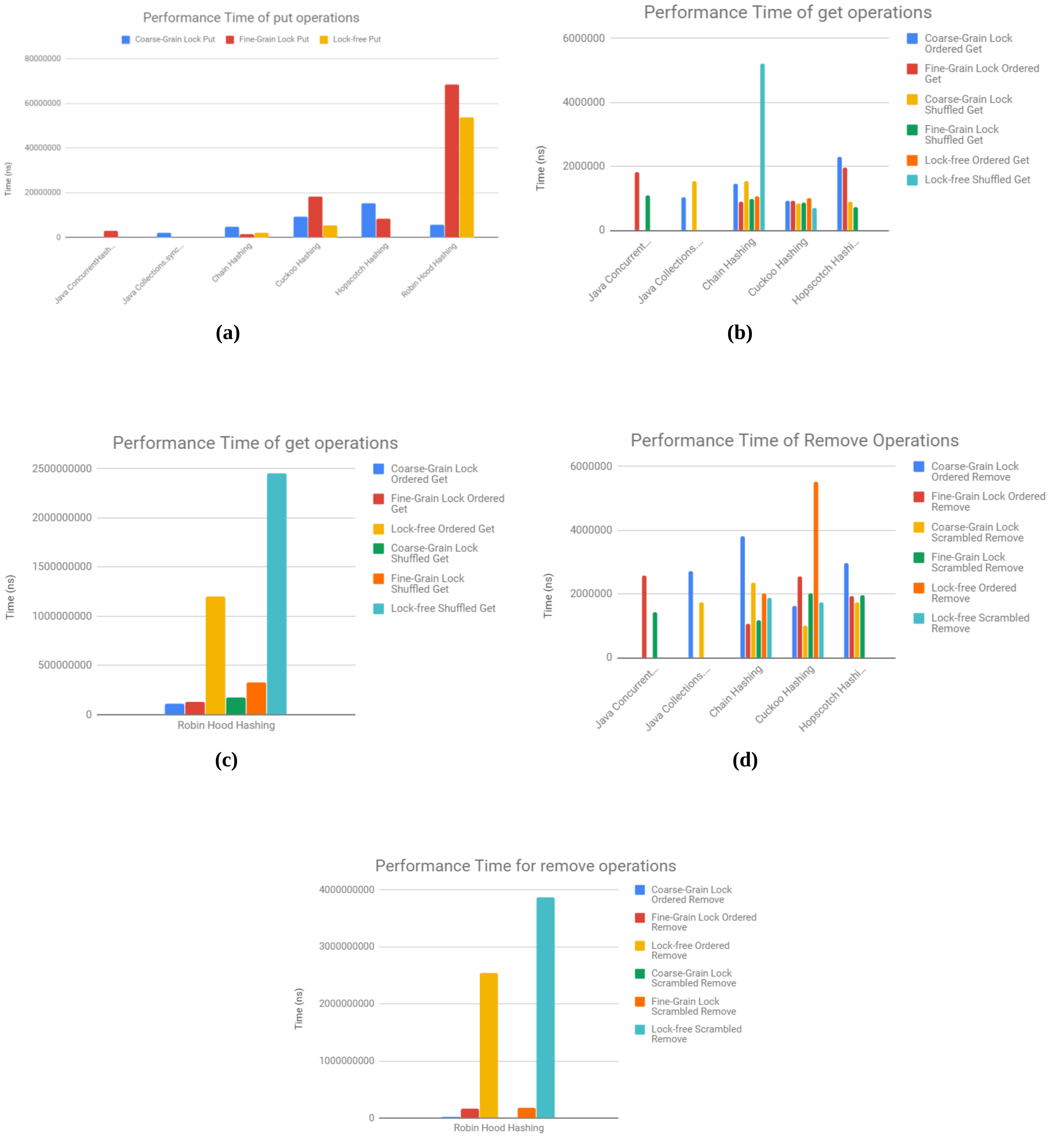
III. Design Alternatives

For the fine-grained and lock-free concurrent chain hashing algorithm, we had to make the decision to synchronize the entire chain at the key level or just synchronize the last entry of that chain. We decided to do the synchronization at the last entry of the chain to improve the performance as much as possible by reducing the potential wait time of other threads that need to work that key.

For fine-grained cuckoo and hopscotch hashing, tables have been shown to do well under lock-striping rather than locking each hash table entry. Our team considered this in our implementation but decided to use locks on a per-entry basis in order to better show the performance of true fine-grained synchronization.

IV. Performance Results and Evaluation

To evaluate our implementations, we wanted to test their performance in the most common operations a programmer would do on a hashtable: put, get, remove. For each operation besides put, we performed the operation twice. Once with a sorted input array of the first 3000 integers, and again with a scrambled array of the first 3000 positive integers. We ran each implementation in parallel using 3 threads. Figure 2 indicates the average performance results of our various concurrent hashing functions. Each entry indicates how long it took our implementation to perform that operation on 3000 integers in nanoseconds.



(a)

(b)

(c)

(d)

(e)

Figure 2: Performance Times, in nanoseconds, of our Implementation of Concurrent Algorithms. (a) Performance of put() for all implementations (b) Performance of get() for the two Java thread-safe hashmaps, chain hashing, cuckoo hashing, and hopscotch hashing (c) Performance of get() for robin hood hashing (d) Performance of remove() for the two Java thread-safe hashmaps, chain hashing, cuckoo hashing, and hopscotch hashing (e) Performance of remove() for robin hood hashing

Of all of the different hashing implementations, it is clear that cuckoo and hopscotch hashing perform the most poorly on "put" operations. Both methods can require an almost linear search of the table during insertion, and insertion can result in a costly "rehash" operation if the number locations a method searches through increases above a certain amount. Interestingly, cuckoo hashing seems to perform worse under fine-grained locking than coarse-grained locking. As expected, fine-grained hopscotch hashing performs better than coarse-grained hopscotch hashing.

Still, the cost in time to insert a value using cuckoo or hopscotch hashing is offset significantly by the time required to "get" a value in either table. Both fine-grained and coarse-grained cuckoo hashing performs particularly well during "get" operations, and when the "get" operations are random, the 3000 "get" operations can be performed more quickly on the lock-free cuckoo hash table than any of the other implementations. Interestingly, hopscotch hashing performs significantly better under random "get" operations than when entries are looked up in order. This difference is likely a result of the contention for nearby locks in the ordered "get" operation. Given this knowledge of the workload, a different hash function could have been chosen to better distribute nearby integers.

Looking at the performance results, we can observe that the Robin Hood hashing algorithm, especially in the case of the Lock-free implementation, has the worst performance out of all the tested algorithm. This poor performance is due to the fact that with Robin Hood Hashing if the hashtable is tightly packed, which is the case of our implementation of this algorithm, collisions occur more frequently and in the worst case-scenario the hashtable essentially becomes an array. While this doesn't impact the timing of the "put" operation too drastically, in the case of the "get" and "remove" the performance drops considerably since those two operations involve looking for the entry.

V. Conclusion

Through this paper, we presented an evaluation of our concurrent implementations of hashing algorithms. We described the project and the hashing algorithms that are the basis of our implementations. In our evaluation, we presented a timing performance of our implementations and compared them to existing Java concurrent implementations as a base line.

References

[1] *Hashing with Chains*. Available at: http://opendatastructures.org/ods-cpp/5_1_Hashing_with_Chaining.html

[2] *Cuckoo Hashing-Worst Case O(1) Lookup!* Available at: https://www.geeksforgeeks.org/cuckoo-hashing/

[3] *Hopscotch hashing*. Available at: https://tessil.github.io/2016/08/29/hopscotch-hashing.html

[4] *How does Java HashMap or JavaLinkedHashMap handle collisions?* Available at: https://javarevisited.blogspot.com/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html

[5] *Robin Hood Hashing on the JVM*. Available at: http://norswap.com/robin-hood-hashing-jvm/

Appendix

Source Code:
https://github.com/JCai2017/ConcurrentHashTable