Implementation of a Non-Blocking Hashtable using C++11

Timothy Flowers, Austin Lasher, and Joseph Maag

***Abstract*—In the paper Non-blocking Hashtables with Open Addressing, Chris Purcell and Tim Harris propose a non-blocking hashtable that utilizes buckets, probe bounds, and open addressing for standard hashtable functionality and achieves non-blocking behavior with the use of atomic lock-free operations, custom word types, and additional states. We implemented the proposed hastable model as a C++11 class utilizing C++11 features and libraries. We maintained lock-freedom with the use of the C++11 atomic library to implement hastable operations that can be performed concurrently. Our implementation was tested over a series of trials and resulted in slower performance compared to our sequential implementation.**

I. Introduction

In this paper we will describe a sequential version of a non-blocking hashtable described by Chris Purcell and Tim Harris in their paper Non-blocking Hashtables with Open Addressing. We implemented the data structure a a C++11 class and performed a series of tests to assess its performance. We will first in Section 1 describe the hashtable proposed in the original paper. Section 2 will describe the details of our C++11 implementation. Lastly, we will describe our performance tests and analyze the results to give an overall view of the efficiency of the concurrent version of the described hashtable and compare our results to our previous sequential version.

II. Algorithm Description

The non-blocking implementation of our hashtable contains three operations: Insert, Delete, and Contains. The function of these operations is described below, as well as a description of the necessary changes for a Non-Blocking implementation.

The non-blocking hashtable utilizes open addressing to resolve conflicts. Each bucket has a bound number that is used as an upper bound on how many buckets that need to be searched during a probing sequence when looking for a key to remove, add, or find. The hashtable described implements this in a non-blocking fashion so any number of threads can be inserting or deleting with progress and accuracy guaranteed. Keeping the bound values correct can be difficult in a concurrent environment. A thread that adds or removes a value attempts to raise or lower the bounds value at the hashed index collision position. At any time during this process, the thread can be paused by the scheduler while another 2nd thread could also add or remove a value. If that 2nd thread finishes, and the original thread resumes, the bound value it sets could now be incorrect.

To alleviate this, a scanning bit is packed into the word that represents the bound value. The bit and bound value can then be accessed atomically with a compare-and-swap operation in one atomic instruction. When a thread lowers a bound value, it sets the scanning bit value to 'true' when it begins scanning bound positions. When finished lowering the bounds, it checks if the scanning bit is still true at the hashed index position. If not, that means that the scanning phase was disrupted, so the process is repeated. This allows bounds to be changed atomically with the correct and accurate value.

When used in parallel, keys could be duplicated as a result of concurrent insertions and deletions. The described hashtable avoids this by assigning a state to each bound position. An empty and member flag can be used to declare that a bucket is empty or occupied, and a bucket that is in the insertion process can be in an inserting state. Before the insertion process and while raising its bound,it is set to a visible state to specify it is not yet in the process of inserting. During its probing sequence for the key, it checks the states of the other buckets. If it finds an already inserted member of the same value, it can stop inserting and set it to a colliding state. If it finds another bucket in an inserting state, it can change that bucket's state to a busy state, signalling it to stall temporarily. Setting a state is done with a swap atomic operation so it won't conflict if another thread changes the state.

To allow lock-freedom and to prevent threads from obstructing each other, a version count value is stored at each bucket to keep track of possible alterations during an insertion process. This value is stored in the same word as the state, allowing for for an atomic single instruction to obtain and set the values which allows threads to safely assist each other concurrently.

A. Insert

Insertion begins with the hashed version of the key to be inserted. The buckets are searched with an increasing number of probing jumps to find an empty bucket (or it will find that the hashtable is full, in which case it returns false). Once an empty bucket is found with a particular number of probe jumps, the bucket state at that position is set to a visible state, and the bound is raised if necessary. The state is then set to an inserting state. It then traverses the probing sequence to assist any other buckets that may be in an inserting process. If any buckets (or threads) are found to be inserting the same value, the original thread's bucket is set to a collided state, and it aborts the insertion by lowering the bound and setting itself to empty.

B. Remove

The removes operation begins by obtaining the hash value of the key to be removed. It then obtains the current value of

the upper probe bound. It then begins the process of looping through each possible collision point through to the probe bound. If it finds a bucket whose key matches the desired key and the bucket is in the member state it will then compare and swap the state to BUSY. If the compare and swap was successful it will lower the probe bounds for the bucket, increment the version count for the bucket, and set its state to EMPTY. If the operation was unsuccessful it will continue searching along the collision path for a match for the key. If the key is successfully removed from the table the function returns true. Otherwise it returns false.

C. Contains

The contains operation begins by obtaining the hash value of the key which is being searched for. It then obtains the current value of the upper probe bound. It then searches every bucket along the collision path through to the probe bound. If it finds a bucket that is in the member state and whose key matches the desired key it returns true. Otherwise it returns false.

III. Implementation

Our C++11 implementation of the lock-free hashtable data structure accurately models the structure described by Chris Purcell and Tim Harris by utilizing C++11 features and libraries, including its atomic library.

A. C++11

The hashtable structure is encapsulated in a C++ class called NBHashTable with a standard C++ class source file NBHashTable.cpp and header file NBHashTable.h. It uses two additional classes, ProbeBound and VersionState, which have their own respective source and header files. C++11 was chosen as our implementation language so we could utilize the thread and atomic library that is part of the C++11 standard library. NBHashTable utilizes std::atomic_int types and compare-and-swap functions (compare_exchange_strong) from the atomic library for atomic access to the hashtable bucket information (bucket bound, scanning state, state, and version) for guaranteed thread safety and implementation of lock-free algorithms. std::thread objects are used in our main test classes to spawn threads and have them interact with a NBHashTable object.

B. Custom Internal Data Types

As discussed earlier, every bucket position needs several pieces of data: the key value at that bucket, the bound value, the scanning bit value, the version value, and the bucket state value. As previously mentioned, the version and state are packed within one word. In our case this was an std:: atomic_int value. However for organizational purposes, we subclassed std:: atomic_int into a VersionState class which has convenient static methods to extract the version number (an int) and scanning bit state (a bool) from an int word. An enum for all 6 states (busy, member, inserting, empty, collided, visible) is provided in the VersionState header file to map directly to integers. Because these two values are still packed into one atomic word, setting and loading the values is still atomic and compare-and-swap operations on a VersionState value is lock-free. A bucket value is represented as a C++ struct BucketT which contains a pointer to a VersionState and a key variable of type NBType, which is simply a typedef int. All buckets are in a BucketT array from which every buckets respective struct data can be accessed. Initially, all BucketT objects are set to a VersionState object with a version 0 and a state of VersionState::State::EMPTY, and a key value of the preprocessor macro EMPTY FLAG defined in the NBHashTable header file. In our implementation it is set to -1 to represent an empty bucket.

As discussed earlier, the probe value and the bound value are packed into one word. Similar to VersionState, it has been abstracted into a class, for organization, called ProbeBound which is an std::atomic_int subclass that has convenient static functions for extracting and creating new ProbeBounds from a scanning bit value (a bool) and a bound value (an int). Just like in VersionState, atomic and lock-free functions can still be performed on a ProbeBound like any other atomic integer value. Each bucket has a ProbeBound to hold its bound value and scanning status, and the ProbeBounds for all buckets are kept in a ProbeBound array in NBHashTable. ProbeBound values are initialized to 0.

To abstract values from VersionState and ProbeBound atomic_int words, each class has a set of methods that take an integer word as an input, and output the needed values by performing bitwise operations to extract them. For example, to find whether the scanning bit is set in a ProbeBound (called pb for example), you can use the ProbeBounds int value with the atomic load() function and then call the respective ProbeBound static function: ProbeBound::isScanning(pb.load()), which will return a bool representing whether the bucket bound is in scanning mode or not. VersionState provides similar static functions to get the version value and state value. Both classes also provide constructors to make new atomic integer words with given values (such as a version integer and a state VersionState::State enum value to make a new VersionState which contains an integer packed with both values).

C. Methods

NBHashTable has three public methods that are used primarily to interact with it: remove, insert, and contains. All have a single argument of NBType, which is is defined in the header file as a typedef integer type. All arguments are expected to be greater than or equal to 0.

The implementation for these three methods all start with hashing the input value to make an associated index that is within the range of this buckets and bounds array. This hash function is simply (value These methods interact with the BucketT array and ProbeBound array to interact with the hashtable values. To maintain lock freedom and atomicity, a compare-and-swap function commonly used throughout these public methods (as well as private methods) to change the value of a VersionState or a ProbeBound after verifying its value. Usually this is done to make sure no other thread has

changed a value in the middle of the process, such as in the public insert method and the private conditionallyLowerBound method. For this we use the compare_exchange_strong method from the atomic C++11 library to perform look-free compare and swaps of ProbeBounds and VersionStates.

There are several private methods to help perform NBHashTable operations. conditionallyLowerBound is a probing process used to lower the bound at a particular bucket value. It performs the probing process and atomically uses the scanning value in each buckets VersionState it encounters for lock-free interaction. conditionallyRaiseBound raises a buckets bound, which is used when inserting a value. It uses the lock-free compare_exchange_strong to make sure the new ProbeBound value is set, and that the original value is the same as it was at the beginning at the process. If it has changed, it repeats the probing process to find its new bound value.

ProbeBound and VersionState values are read atomically and without locking. The values stored in their respective int word are extracted with public static methods within each of their respective classes, such as VersionString::getVersion() and ProbeBound::getBound() (both of which take an integer argument).

D. Lock-Freedom

The structure uses bit stealing in order to implement the mechanisms used for the progress guarantee and the correctness condition. Bit stealing is used in two instances. In the first instance it is used in order to steal three bits from the version number in order to store the current state of the bucket is in. The states include: busy, member, inserting, empty, collided, and visible. When in the busy state, the bucket is currently being acted on by the insert or remove operation but its effects are not observable in any way. When in the member state, the bucket's data is both valid and not currently being acted on by any operation. When in the inserting state, the bucket's key is valid but the collisions with the bucket may have not been resolved yet. When in the empty state, the bucket is not being acted upon by any operation but the bucket's key is not valid. When in the visible state, the bucket's key value is valid but its probe bound are not currently valid. In the second instance a single bit is stolen from the probe bound in order to store the scanning state for the particular bucket.

This bit stealing becomes useful when the compare and swap operations are used to synchronize operations across threads because it allows the program to compare multiple values atomically and ensure that another thread doesn't act on the data before another thread is able to complete its current operation. By ensuring that the threads are able to achieve complex state based synchronization, correctness is ensured. In this particular implementation the insert and remove operations both use the compare and swap operation in order to ensure that the version of the current bucket is unchanged and that the state of the of the bucket is what is expected before changing the state of the bucket to the next one within the given process the bucket is involved in. The bit stealing is also used by the conditionally lower bound operation in order to set the scanning bit at the same time as the probe bound in order to be sure that another lower bound operation does not conflict with the same bucket.

Through the use of these bit stealing operations in conjunction with the compare and swap operation, a lock free progress guarantee is achieved. In the event of failed compare and swap operations the operation is reattempted until the compare and swap succeeds. If a particular operation is interrupted it is because another operation succeeded, therefore at least one thread was able to complete its operation.

IV. Testing and Performance

To compare our algorithm to the original description, we set up several testing suites that implemented our NBHashTable class. In the previous assignment, we focused on both correctness, and execution time. This time around, as we have added some additional structures to support non-sequential loads, we required several new tests in addition to the previous tests.

The first new test is our Atomic Types test file, "atomictypes.cpp". This program allowed us to create test cases for our ProbeBound and VersionState classes. As the paper showed, we used a bit-stealing technique to combine the probe bound and scanning bit into a single memory word, so a single compare-and-swap operation could update this value. The same process needed to be done for the Version and bucket state variables, except the bucket state required three bits as there were six possible states.

The second new test file is titled "instructions.cpp", and allows us to input exact numbers we wanted inserted and deleted. This way, we can set up even more specific tests and make sure that each function was working appropriately. These are input using standard input, so files can be piped as input to the program.

A. Correctness

Correctness is difficult to measure for all cases. Unlike execution time, it would be difficult to create a testing suite to measure correctness that randomly generates numbers. It would randomly have to generate insertions and deletions of numbers, and from that compare it to a "correct" output, that it would also have to generate. How can we be sure that the output it created was actually correct for every case? There is no way to be sure, as there could be a fault in our procedure that generates a "correct" output. A false positive, in this case, would be likely.

Our approach to measuring the correctness of the algorithm, instead, was to create several test cases. The idea is that we can create test cases, and compare the actual result to an expected result we retrieved on paper. With this method, we can create many different tricky test cases, and see how they perform individually. From each tricky test case, we were thereby able to refine our implementation to be certain of its correctness, and kept refining different portions of code until we were certain it was correct.

We have included our sample test cases in our initial submission document, under 'correctness1.cpp' and 'correctness2.cpp'. These can be built using our makefile with the target "make correctness". This will make two corresponding

executables, correctness1 and correctness2 under the build/ directory. These can be run to show our testing methodology, and the proof of our algorithm can be visually seen.

B. Execution time

Execution time is important to test because it allows us to compare the specific operations of the algorithm and determine how it should be used most efficiently. It also allows us to compare our implementation to the expected time from the original paper, and potentially see the limits of our currently sequential solution.

Our goal is to measure execution for a potentially random set of data in bulk, and measure the approximate execution time depending on the number of threads. We also hope to visually see a correlation between the number of operations performed and the resulting execution time for those set of operations. To do this, we created a C++ program (executiontime.cpp) that we can run multiple times with different parameters, and will show the average execution time on the given parameters.

With this goal in mind, we approached the program by spawning some number of threads, and performed 500,000 operations on each of those threads. These were either insert() operations, delete() operations, or contains() operations. The distribution of each of these was modified into six different cases, shown in Table 1 below.

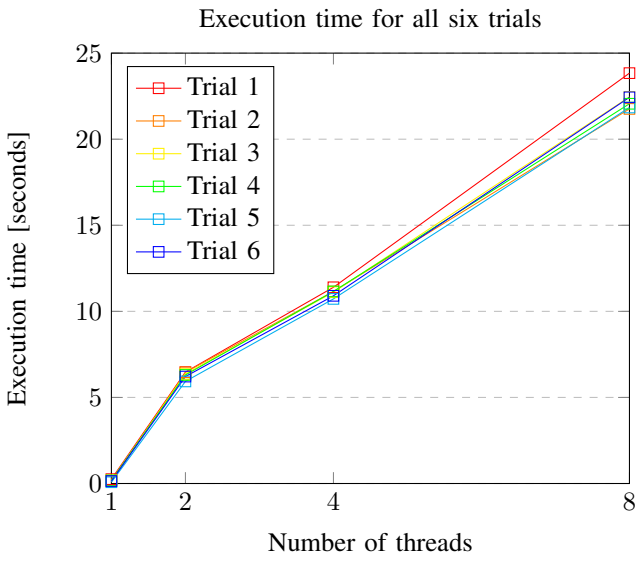
Trial #	insert()	delete()	contains()
1	75%	25%	0%
2	25%	75%	0%
3	50%	50%	0%
4	45%	10%	45%
5	10%	45%	45%
6	33%	33%	34%

TABLE I
DISTRIBUTION OF OPERATIONS FOR EACH TRIAL.

Our C++ program takes four command-line arguments. The first is the integer number of threads. If no arguments are specified, the default distribution is used, and the program will ask how many threads to perform on. The first argument is the number of threads. Next are three arguments: three different percent chances for our three different operations to be performed. The first corresponds to the weighted probability of an insert operation, the second a deletion, and lastly our look-up operation (contains.)

The program will automatically run each operation with a random integer between 1 and 100, and perform these operations on a hashtable with only 50 slots. These options are customizable inside the source C++ file as #define constants.

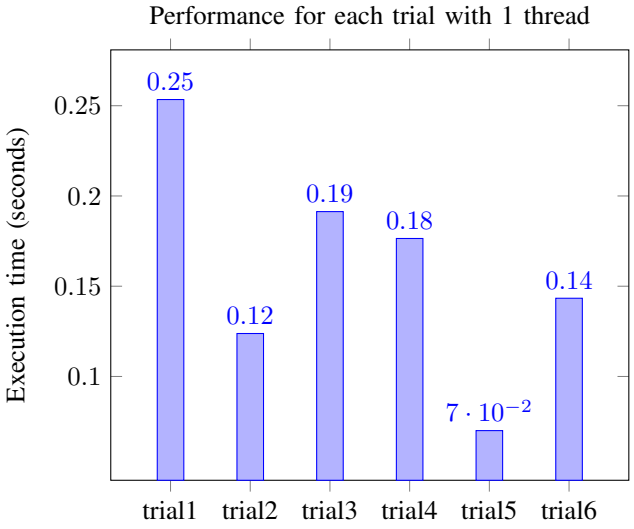
Because we want our test results to reflect an accurate average timing for normal use, we decided to each of our tests multiple times, timing each one, and average the results together. This was done directly inside the coded file. The number of average trials is configurable as well, and the program will print out the timing for each trial. By default, we have configured this to time each trial four times, and print the average of each measurement.

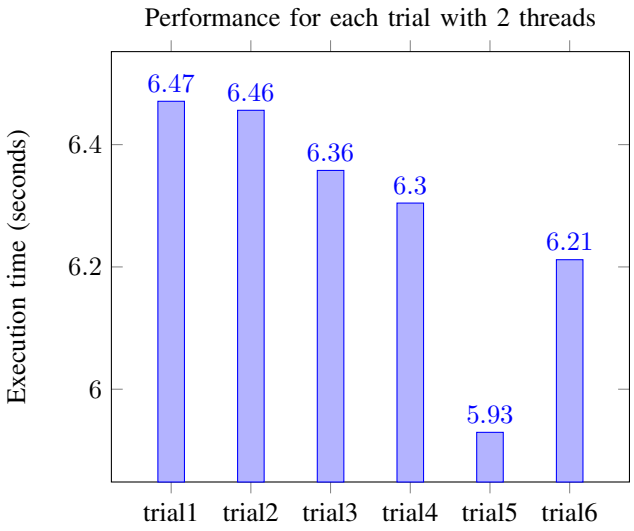


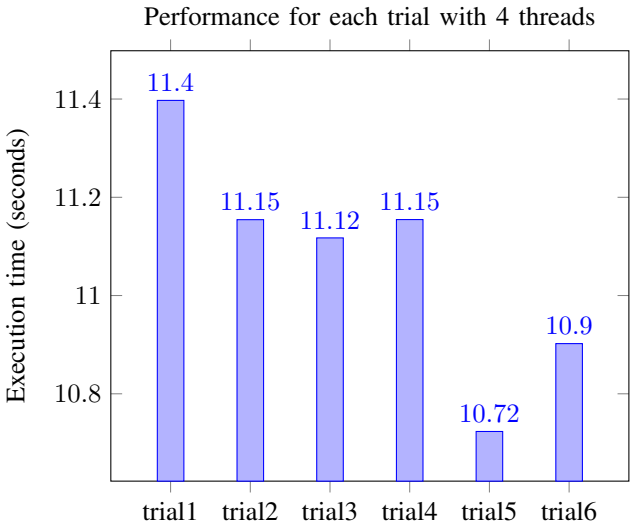
Using this program, we were able to create a huge GNU Makefile that automated our tests, and piped the output to a testing directory. The very same Makefile is available in the included package, and can be run using the target ”make executionTimeTests”, as described in the README file.

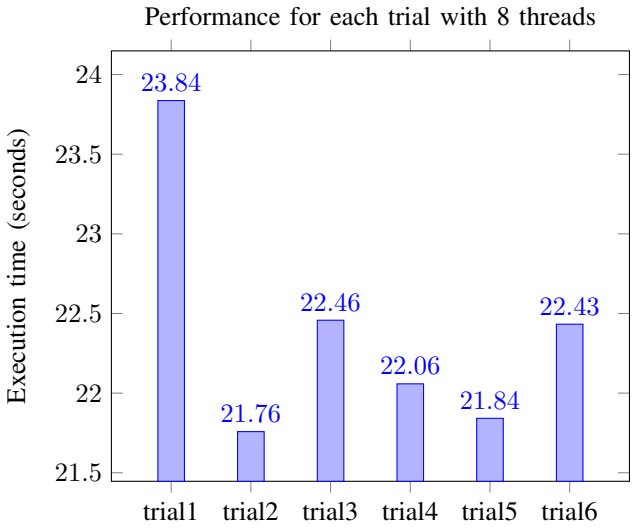
C. Results

The overall results of each of our execution time trials can be shown in a single graph above. We’ve included this graph as it was asked for in the assignment description. However, we felt that it was more useful to compare each trial with the same number of threads to each-other side-by-side. This allows us to view the results in a more meaningful manner. Each of these graphs have been included below.









For each graph, we can see that trial 5 took considerably less time for each test with all the different thread possibilities. Consulting Table 1, we can see that thread 5 has the least insertion likelihood. A low time is also noticeable for trial 2 on the 1-thread and 8-thread graphs. Our interpretation is that insertion into our table is the most expensive operation, at least in this initial sequential implementation. Judging by the high results of trials without any contains operations, we feel this is evidence of contains() being the least expensive operation.

It will be interesting to see the affect of the next step in this project: implementing the non-sequential data structure. We expect the final non-blocking hashtable, as originally described, should severely decrease the execution time of our results. The nice thing about our execution C++ code is that they implement the class in a generic fashion, so that we can modify the hashtable code and test against the exact same execution time code.

V. Analysis and Conclusion

The non-blocking hashtable proposed by Chris Purcell and Tim Harris uses a number of atomic operations, lock-free operations, and specialized data types to achieve non-blocking functionality. Our implementation followed the provided algorithms precisely with slight modifications for the use of C++11.

References

[1] C. Purcell and T. Harris, *Non-blocking Hashtables with Open Addressing*. Springer-Verlag, Berlin, Heidelberg, 2005.