

CSC 464 – Project

A Lock-Free Hash Table

*Michael Windels
December 13, 2018
V00854091*

*For Dr. Yvonne Coady's CSC 464 (Fall 2018) class.
Repository: <https://github.com/MWindels/uvic-csc464-project>*

1 The Problem

When implementing a thread-safe data structure, one's choice of synchronization mechanism can profoundly affect how performant it is. Coarse-grained locking, for example, has a serious impact on performance. A shared data structure with a coarse-grained lock has a single lock over the entire data structure. That singular mutex serves as a bottleneck which every thread must pass through if they want to access or mutate the data structure. This makes all accesses and mutations to the data structure sequential, and prevents concurrent use of the data structure. While specialized mutexes like the readers-writers mutex can introduce concurrency for readers, writers must still exclude all other threads from observing the data structure. Despite being simple to implement, coarse-grained solutions to concurrency control simply cannot scale due to the singular mutex acting as a bottleneck for all threads.

While concerns about concurrency control can apply to any data structure shared amongst multiple threads, the focus of this report will be on hash tables. Hash tables are our focus because they are a data structure which (if implemented well) should be easily accessed and mutated. Furthermore, hash tables are used in a wide variety of different applications. Some applications may be read-heavy, while others may be write-heavy. A decent implementation of a thread-safe hash table should take into account the different circumstances in which the table might be used. In any case, a sensible implementation of a thread-safe hash table should avoid coarse-grain locking.

2 The Solutions

Rather than controlling concurrency with such draconian means as coarse-grain locking, let us consider doing exactly the opposite. The solutions we will consider refine the granularity of their synchronization mechanisms down to the finest they can possibly be. Our solutions will use atomic operations as their synchronization mechanisms. Our goal is lock-freedom in a hash table.

2.1 Other's Solutions

Implementations of lock-free hash tables are hardly new. While at IBM, Maged M. Michael devised an implementation of a thread-safe lock-free hash table in a paper published in 2002 [1]. Among other things, this table used chaining as a means of resolving hash collisions. Chaining removed the need to consider resizing the hash table. The lock-free data structure at the core of this implementation is the singly-linked list, not necessarily the table itself. The singly-linked list is a relatively simple data structure to implement using only atomic operations (especially when compared to the doubly-linked list).

More recently in a blog article from 2016, Jeff Preshing detailed his own implementation of a lock-free hash table [2]. Unlike Maged Michael's table, Preshing's table uses linear probing to resolve collisions. While Michael's table is able to delegate the maintenance of consistency to the internal (lock-free) singly-linked lists, Preshing's lock-free table must take on greater responsibility for ensuring consistency. Linear probing also necessitates the ability to resize the table. While get operations and (non-resizing) set operations use atomic operations to access or mutate key-value pairs, Preshing opted to use a lock to control access to the new table being created.

In 2015, Shlomi Steinberg also implemented a lock-free (even wait-free, ostensibly) hash table [3]. Steinberg's implementation highlights the importance of how one reclaims memory in a lock-free data structure. Their double-counting reference counter is a simple but effective way of managing memory in data structure that is allocating and freeing memory very often.

2.2 My Solution

Though the other implementations were all very insightful, my own implementation does not wholly emulate any of them. Instead, my implementation is a fusion of the other implementations combined with some attributes of my own making.

The lock-free hash table which I implemented uses linear probing to resolve collisions. However, the table does not lock to resize like Preshing's table. Instead, the table stores a doubly-counted reference to the next table. The doubly-counted reference draws from Steinberg's implementation, though they use their references for different purposes. If a set operation is attempted (with a novel key) and the last table is full, a new table is created and the last table links to it. The hash table is therefore not a single table, but a chain of tables. It is similar to a singly-linked list, but tables can only be added to the end, and they can never be removed. This vastly simplifies the (atomic) addition of new tables to the chain. The length of the chain grows logarithmically with the number of unique keys inserted into the table. This is due to the fact that the size of a new table is always twice the previous size. While resize operations effectively become unnecessary, the get and set operations on the table become logarithmic, not constant. Figure 1 illustrates what the hash table looks like abstractly.

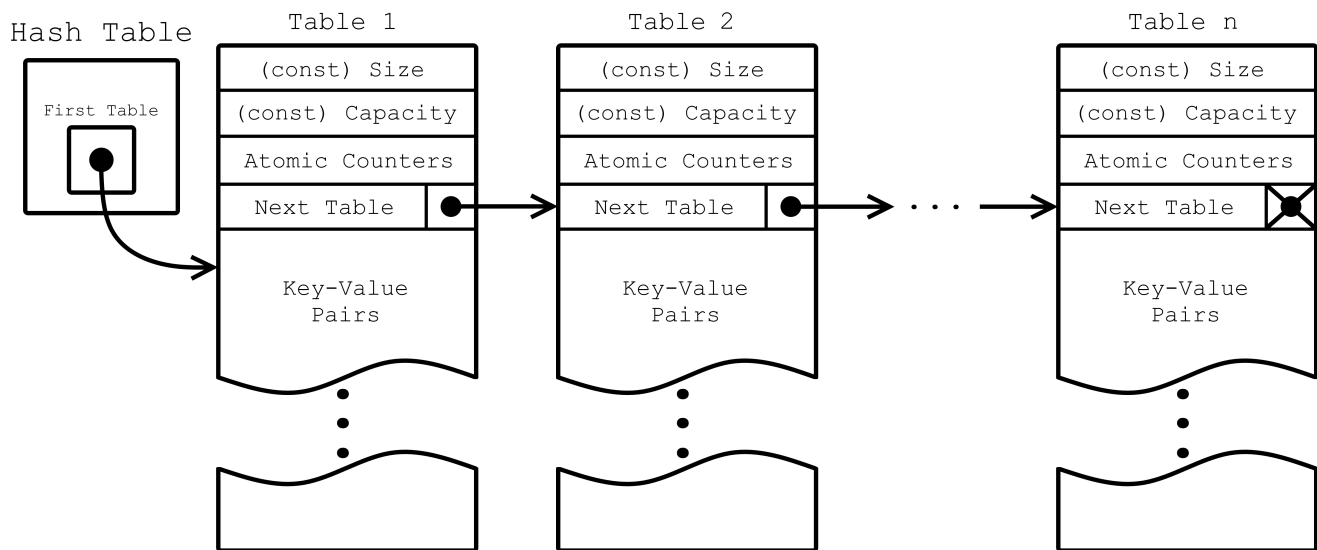


Figure 1 – The structure of the lock-free hash table.

Every key-value pair in each table (along the chain) uses doubly-counted reference counters for storage. This allows us to atomically obtain and replace them. The doubly-counted reference counters also force data which is still being observed to persist on the heap even if it is otherwise totally inaccessible. This means that new key-value pairs will never accidentally re-use memory belonging to old key-value pairs until no more threads are observing the memory.

The solution is lock-free in the sense that it does not use mutexes, but it is also lock-free in the technical sense. Technically, lock-free algorithms are a class of non-blocking algorithms. Lock-freedom is the guarantee that at any time, at least one thread will be making forward progress. In our lock-free hash table, other threads may have to spin in a compare-and-swap loop if they are operating on the same data. Unless there is extremely high contention on a single key, no thread should have to spin for long.

Because C++ allows for greater control over atomic variables than Go (another language considered for this project), the lock-free hash table is written in C++. The implementation can be found at the following link:

https://github.com/MWindels/uvic-csc464-project/blob/master/src/lib/lockfree/hash_table.hpp

3 The Evaluation

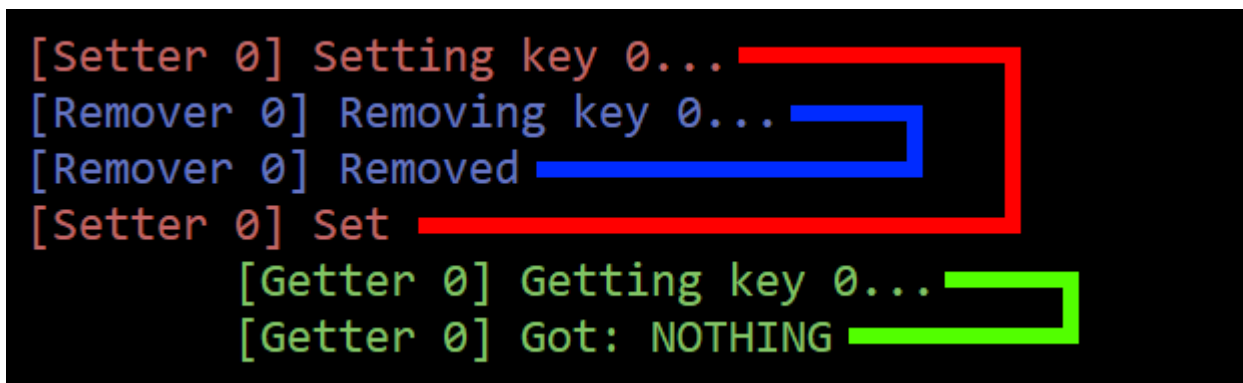
The `src/tst` directory in the repository contains three little test suites. The two which test the hash table are the `table_tester` and `table_timer` files. The `table_tester` and `table_timer` files can be found at the first and second links respectively:

https://github.com/MWindels/uvic-csc464-project/blob/master/src/tst/table_tester.cpp

https://github.com/MWindels/uvic-csc464-project/blob/master/src/tst/table_timer.cpp

3.1 Correctness

The `table_tester` file evaluates the correctness of the lock-free hash table by spinning off an arbitrary number of getters, setters, and removers. Correctness is evaluated by examining the output from getter threads and comparing it to output generated by concurrent set and remove operations. Some typical output of `table_tester` can be found in Figure 2.



```
[Setter 0] Setting key 0...
[Remover 0] Removing key 0...
[Remover 0] Removed
[Setter 0] Set
      [Getter 0] Getting key 0...
      [Getter 0] Got: NOTHING
```

Figure 2 – Typical output from the `table_tester` program.

The coloured lines in Figure 2 denote the intervals in which the listed operations could be linearized. In the context of concurrent programming, a linearization of a series of operations is a sequence of linearization points. The linearization point of some operation is the unique point in time

at which the operation is said to have happened. In the case of Figure 2, it is valid that the getter observed nothing at key zero. The linearization $[\text{Set Zero}] \rightarrow [\text{Remove Zero}] \rightarrow [\text{Get Zero}]$ would yield this answer.

Problematically, `table_tester` does not allow us to reliably test the table's corner cases. In particular, there is a pathological case where it is possible for the table to end up with duplicate keys. The case occurs in the event that two (or more) set operations attempt to concurrently insert the same key when the last table is one element away from being full. One set operation may reserve the last spot in the last table, but the other set operation may not observe the value in the table. Since the last spot in the table is reserved, the other set operation must create a new table and insert there. This race is resolved by having all get operations always use the last value they read. If we consider the set operation's point of linearization to be when it reserves a spot for insertion, or when it falls off the end of the table chain (after updating previous spots), then we can still guarantee determinism. Hence, the race is resolved.

3.2 Performance

To evaluate how performant my implementation of a lock-free hash table is, a second thread-safe hash table was implemented which uses coarse-grained locking to manage concurrency. In the interest of fairness, the locking table uses a readers-writer mutex so at least get operations can happen concurrently. If a regular mutex were used, the table would perform poorly no matter what operations were being performed on it. The locking hash table can be found at the following link:

https://github.com/MWindels/uvic-csc464-project/blob/master/src/lib/locking/hash_table.hpp

The `table_timer` file is the testing program which evaluates the performance of the lock-free hash table. The program works by spinning off a number of getter and setter threads which all perform some common number of operations each. A monotonic clock is used to time the duration of each get and set operation. Once the getters and setters have joined the main thread, the average time taken for every get and set operation is outputted along with the standard deviation of the operations. The testing program can be used to test both the lock-free table, and the locking table.

Results from the `table_timer` program were mixed. When ninety-nine percent of threads were performing get operations, get and set operations in both tables performed extremely well (in the range of microseconds). This exceptional performance is likely due to the fact that there were very few set operations, and both tables contained only ten thousand key-value pairs. When the percentage of get threads decreased to ninety percent, the locking table suddenly appeared about an order of magnitude faster than the lock-free table (at least for get operations). However, once the percentage of get-threads dropped below fifty percent, the locking table began performing much more poorly than the lock-free table. At one percent get-threads, set operations on the lock-free table only took two milliseconds on average, while they took six and half on the locking table. The difference between get operations on the lock-free and locking tables was even more stark. Figures 3 and 4 demonstrate the average time taken to perform a get or a set with varying amounts of get-threads and set-threads. Figures 3 and 4 are derived from tests in which the threads performed ten thousand operations each.

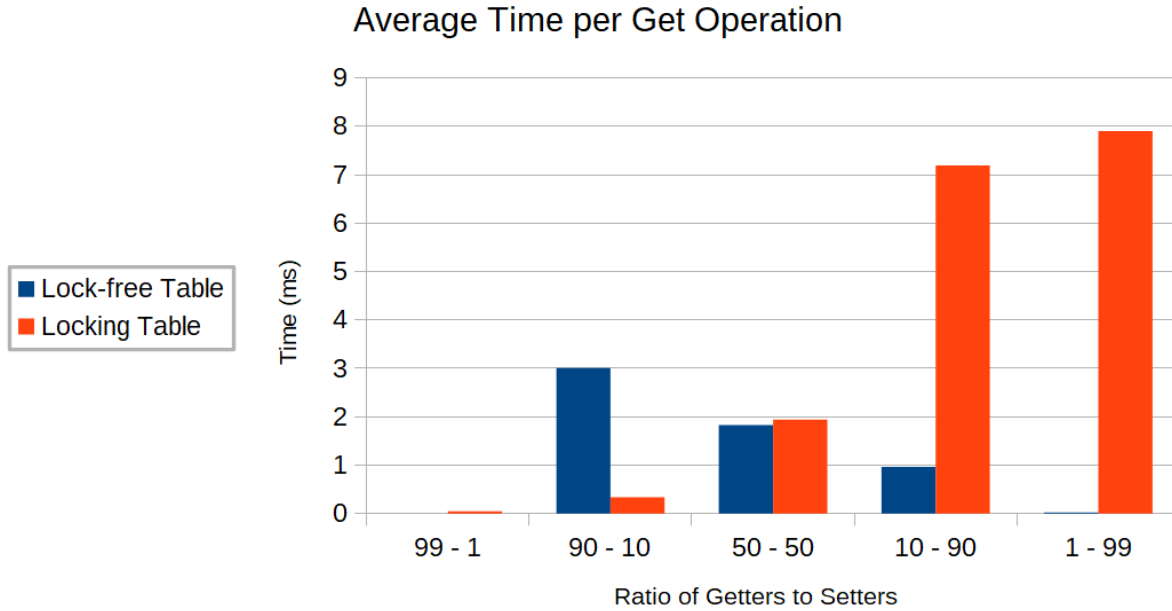


Figure 3 – The average time taken for get operations.

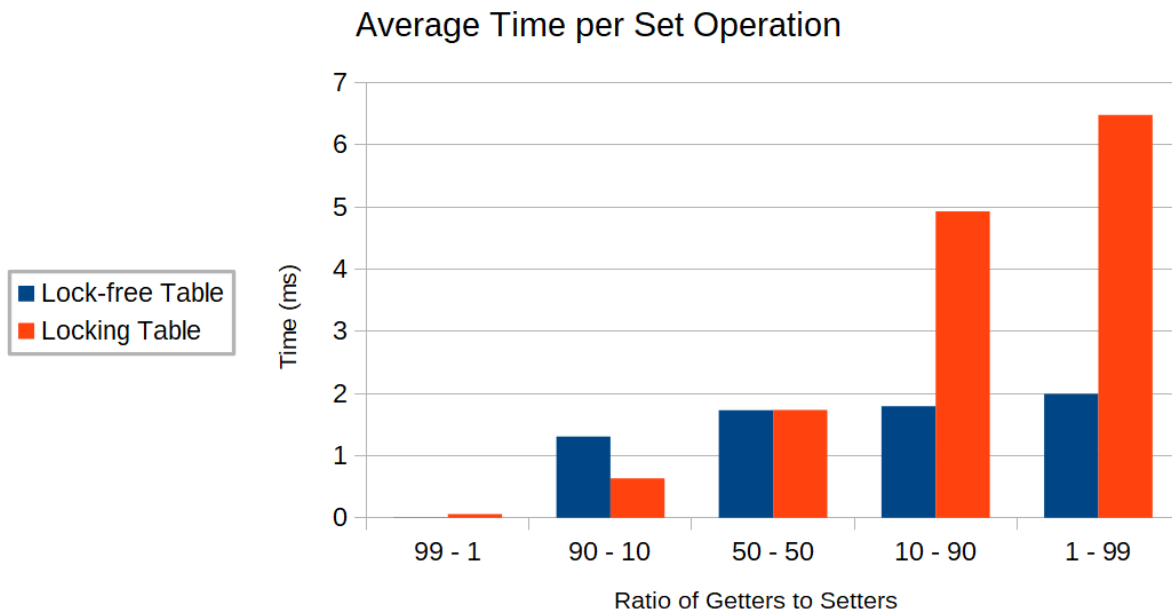


Figure 4 – The average time taken for set operations.

The standard deviations also reveal how unreliable the locking implementation is under heavy set-contention. Because the locking hash table uses only a single table which must periodically be resized, some set operations can take much longer than they would otherwise. Additionally, every set operation which resizes the table must lock out all get-threads and set-threads, increasing the time those threads take to perform their operation. Figures 5 and 6 demonstrate the standard deviations for each of the five tests in Figures 3 and 4.

<i>Standard Deviation of Time Taken for Get Operations</i>					
	<i>Ratio of Getters to Setters</i>				
	<i>99 - 1</i>	<i>90 - 10</i>	<i>50 - 50</i>	<i>10 - 90</i>	<i>1 - 99</i>
<i>Lock-free</i>	0.009 ms	29.225 ms	21.794 ms	14.614 ms	0.825 ms
<i>Locking</i>	1.418 ms	10.606 ms	42.928 ms	124.018 ms	138.531 ms

Figure 5 – The standard deviation of time taken for get operations in both tables.

<i>Standard Deviation of Time Taken for Set Operations</i>					
	<i>Ratio of Getters to Setters</i>				
	<i>99 - 1</i>	<i>90 - 10</i>	<i>50 - 50</i>	<i>10 - 90</i>	<i>1 - 99</i>
<i>Lock-free</i>	0.018 ms	16.142 ms	21.000 ms	22.435 ms	23.017 ms
<i>Locking</i>	2.442 ms	14.556 ms	40.156 ms	92.441 ms	122.334 ms

Figure 6 – The standard deviation of time taken for set operations in both tables.

Despite the get and set operations being logarithmic, not constant, it would appear that under heavy set-contention the lock-free table dramatically outperforms the coarse-grained locking table. Even with a readers-writer mutex, the locking table simply does not offer the same degree of concurrency offered by the lock-free table.

4 Future Work

While my implementation of a lock-free hash table is more performant than a coarse-grain locking hash table when there are a substantial number of concurrent setters, there is still room for improvement.

For this project, I chose not to try resizing the hash table as it filled up. This led to the lock-free hash table looking more like a chain hash tables. Originally, I had plans to implement a method that would consolidate key-value pairs from earlier (smaller) tables into later (larger) tables. This would have the effect of reducing the length of the table chain, thus reducing the amount of time necessary for future get and set operations. Such a method would be especially useful in applications which initially perform a lot of set operations, then perform mostly get operations. Actually implementing such a function was problematic. One must consider what happens when a value from an early table is written ahead to a later table while a set operation (on the same key) happens concurrently. It may be possible that a new value overwrites the old value just after the old value was loaded, but before it was atomically written ahead. In such a case, the set operation may appear to have never happened. Matters are further complicated by the potential for a racing set (described at the end of section 3.1) in which one of the racing threads is the thread moving elements down the table chain. While it may be possible to implement a lock-free method for consolidating key-value pairs into later tables, it is beyond the scope of this project.

Another candidate for future work is relaxing the `memory_order` of the atomic operations used by my lock-free hash table implementation. Because atomic variables are often used to coordinate access to shared data, C++ allows the programmer to specify the `memory_order` of an atomic operation. The `memory_order` of an atomic operation is important for two reasons. First, because compilers and hardware can (and will) reorder reads and writes in your code for efficiency. Second, because memory written to a cache being used by one thread is not necessarily visible to all other threads. An atomic operation's `memory_order` determines three things:

- How reads and writes can be reordered around the atomic operation.
- Which writes from other threads are visible to you after operating on a shared atomic variable.
- Which of your writes are visible to other threads after operation on a shared atomic variable.

C++ supports three categories of atomic operations based on `memory_order`: *relaxed operations*, *acquire loads* matched with *release stores*, and *sequentially consistent operations*. Each of these categories provide different guarantees with respect to reordering and visibility of writes. Sequentially consistent operations ensure that all threads can see the writes performed by other threads after operating on a shared atomic variable. This memory ordering is the strongest, but also the most expensive. By contrast, a relaxed atomic operation provides no guarantees of write visibility, but is also very inexpensive on some architectures like ARMv7. Acquire and release operations occupy a middle ground between sequentially consistent and relaxed operations. Atomic operations which are not sequentially consistent can be extremely difficult to reason about.

In the interest of correctness, all of the atomic operations used by my lock-free hash table are sequentially consistent. However, because the performance tests in section 3.2 were carried out on an x64 machine, the performance impact of sequential consistency was less severe than it might have been otherwise. This is because x64 (and x86) provide strong memory ordering guarantees by default. Despite this, x64's memory ordering guarantees on stores are not quite sequentially consistent, so it is possible that there was some performance impact.

I believe that the `memory_order` of some atomic operations in my lock-free has table could be relaxed to the level of acquire and release, but I am unsure how I could properly test for correctness without sequential consistency. While the lock-free table already performs decently, relaxing the `memory_order` of some operations could yield even better performance.

5 Sources

- [1] M. Michael, *High Performance Dynamic Lock-Free Hash Tables and List-Based Sets (white paper)*, 2002. Available: <http://www.liblfd.org/downloads/white%20papers/%5BHash%5D%20-%20%5BMichael%5D%20-%20High%20Performance%20Dynamic%20Lock-Free%20Hash%20Tables%20and%20List-Based%20Sets.pdf> [Accessed Dec. 13, 2018].
- [2] J. Preshing, “A Resizable Concurrent Map,” *preshing.com*, Feb. 22, 2016. Available: <https://preshing.com/20160222/a-resizable-concurrent-map/> [Accessed Dec. 13, 2018].
- [3] S. Steinberg, “Designing a Lock-Free, Wait-Free Hash Map,” *shlomisteinberg.com*, Sept. 28, 2015. Available: <https://shlomisteinberg.com/2015/09/28/designing-a-lock-free-wait-free-hash-map/> [Accessed Dec. 13, 2018].