Exploration of a Lock-Free Hash Table

Trevor Absher, Martin Dinkov

November 30, 2018

1 Abstract

This paper expands upon an agorithm presented in *Dynamic-Sized Nonblocking Hash Tables* by Liu et. al. A specialization of this algorithm lies in its ability to dynamically resize in both directions. This ability is achieved through a freezable set abstraction that expands upon the standard set operations. A freeze method prevents any thread from editing a given bucket during the hash table resizing process.

We focus on the lock-free version of the hash table, using C++ to implement it. A minor difference lies in the fact that our implementation uses a Linked-List based set. The authors used a single flat array for its benefits regarding cache locality. However, an array-based implementation requires additional bookkeeping to avoid fragmentation. Our lock-free linked list Set implementation is derived from *The Art of Multiprocessor Programming* by Herlihy and Shavit.

The idea of freezing the hash set relies on a resizing operation, dictated by a heuristic policy. The original paper does not cover any preferred heuristics. Thus, we compare the application of various commonly used policies. We also implement a transactional variant of a sequential hash table for the purpose of comparing it to our lock-free concurrent implementation. Our testing for each solution is done across varying hardware architectures in order to explore the solution's portability.

2 Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel Programming;* E.2 [**Data Storage Representations**]: Hash-table Representation

3 Implementation & Architecture

3.1 Freezable Set

The Freezable Set class (FSet) serves as a wrapper layer around the Set class. The FSet object contains *set* and *ok* variables and defines the FSetOp struct. The *ok* variable is a single bit representing whether or not the FSet is frozen. If *ok* is set to false, then *set* becomes immutable and no threads can add or remove an element. Threads can still check if an element is in the frozen set and retrieve it; which is important for the *InitBucket* method of our HashTable, which will repopulate the current head node's buckets with the data from the previous head node.

The *HasMember* method solely exposes the *Contains* call of the underlying *Set* class. The *Invoke* method triggers either an add or remove operation. This method checks if this operation has not already been completed and if it is a valid operation. After those conditions are met, either *Set::Insert()* or *Set::Remove()* is called dependind on the *op.type* variable; which is set to an enum of either **INSERT** or **REMOVE**. The *Freeze* method is simple, yet crucial to the functionality of making the hash table lock free. We made no changes to the Freeze method provided by [1].

A size variable is added to the FSet class as an aide for one of the explored hash table resizing heuristic policies. This variable is only an estimate of the size of the set since it is modified past the linearization point of an *Insert* or *Remove* operation. Implementing a more accurate size query requires use of descriptor objects within the Set class. Descriptors would lead to unneccessary overhead since such size accuracy is not necessary solely for the purpose of determining when to resize.

3.1.1 FSetOp

The FSetOp struct contains an enumeration with two values **INSERT** and **REMOVE** set to true and false respectively. The *key* variable represents the integer that is going to be either added or removed. The *done* bit is an atomic variable that represents if this specific add/remove operation has already been completed. The *resp* bit is an atomic variable that represents if the operation was successful or not. The operation might fail if an add attempts to add a duplicate value, or if a remove attempts to remove a value that is not in the set. These variables are atomic to ensure all threads see the same statuses of these operations. The *GetResponse* method retrieves the *resp* variable of the object.

3.2 Set

The authors of [1] use an array-based set for the core of their bin implementation. However, they do not go into the details of their Set implementation since it is irrelevant to the lock-free hash table algorithm. We use a linked-list based Set class to represent the integer set of the FSet object based on the lock-free version of the Set object given in *The Art of Multiprocessor Programming* [3]. The lock-free Insert and Remove operations provided by this algorithm are necessary to maintain the lock-free property of the hash table. Our chosen Set algorithm also provides a wait-free Contains operation, which is the operation most commonly used in a data structure. The values contained within the Nodes are unique integers and ordered, thus also serving as keys to Node objects in our implementation.

The Set class supports two traversal methods. Set::Find is used for internal traversal within the Set::Insert and Set::Remove calls in order to provide a window of two nodes. The Set::Find method can also plays a part of a lazy removal process. It physically removes any Nodes that have been logically flagged for removal. The logical removal flag is embedded within the first bit of the pointer that links the Node. Bit-stealing is used in order to avoid the overhead of using a descriptor object and to maintain linearizability. The physical deletion is done using a CAS call. If this CAS call fails, then a neighboring Node must have been inserted or removed and the traversal must start over from the first sentinal Node. Such failure denotes that another thread must be making progress.

Set::Contains provides the other traversal method, which always traverses each Node once and linearizes in a deterministic amount of steps. This deterministic quality of Set::Contains guarantees wait-freedom. The Set::Contains method linearizes once it has found a Node that has a value that is NOT less than the value that is being queried. If the first such value found is greater than the value queried, then the value queried is not in the Set since the Nodes are ordered based on their values.

The Set::Remove method obtains a Set::Window of two Nodes using Set::Find based on the value that is to be removed. If the second of these Nodes has a value different from the one to be removed, then the value to be removed does not exist in the Set and the Set::Remove call fails and returns false. Alternatively, if the value to be removed is found, then the corresponding Node is logically flagged for deletion using a CAS call. Similarly to the physical removal in Set::Find, failure of this CAS call denotes that a another thread has made progress by inserting or removing a neighboring Node. In such case of this CAS failure, the Set must be re-traversed from the beginning. A logical deletion is attemped using CAS call following the physical deletion. It is okay for this second CAS to fail as the Node will be physically deleted in the following Set::Find call.

The Set::Insert method also obtains a Set::Window of two Nodes, similarly to Set::Remove. If the value of the latter of the Nodes is equal to value being inserted, then the value already exists and the Set::Insert call fails by returning false. Otherwise, the value is inserted between the two Nodes from the Set::Window object. A new Node is created using the value being inserted. This Node is inserted using a CAS call. If the CAS call happens to fail, it is only due to another call making progress and the Set must be retraversed once again.

The original algorithm provided by Herlihy and Shavit [3] is extended with the implementation of the *Set::Union* and *Set::IntersectionRemainder* methods. Such methods are utilized for the binary resizing capability of the housing hash table class. A Set::Union call is provided with another Set as an argument, which is combined with the Set instance upon which the method is called. The Set::IntersectionRemainder iterates through the Nodes of the Set instance upon which the method was called and picks any Nodes that meet a criteria into a new Set. The criteria is that the value of the Node must have a given remainder when divided by a given divisor. Both of the aforementioned methods do not require thread-safe properties because they are only called once parenting FSet class has been frozen and no further modifications to the Set can be made.

3.3 Hash Table

As most hash tables, this one's public interface supports three methods: *Insert*, *Remove*, and *Contains*. The pseudocode for these methods are specified in [1]. This HashTable implementation can house up to two internal hash tables, which are referred to HNodes in [1]. Having up to two HNodes allows for the unique resizing properties of this lock-free data structure. Below are two critical distictions between the HNodes: The first HNode must always be present and is accessed via a head HNode member pointer. The second HNode can be accessed through the *pred* pointer of the first HNode and is non-existant if that pointer equals NULL. The HNode pointers are atomic variables since they can be accessed by multiple threads. The first bin can have un-initialized (NULL) bins. The second HNode is used to initialize the bins of the first HNode. All bins of the second HNode MUST be initialized and cannot be NULL. A third HNode is not present to initialize the bins of the second HNode.

Set::Resize is an internal call that can be triggerred by an Insert or Remove based on a heuristic policy. The heuristic policy choise can significantly impact performance. Set::Resize should'nt be called too often since the Set::Resize call is expensive. However, Set::Resize should be called often enough so that the bin sizes do not become too large, which would result in expensive traversals. In an effort to avoid the overhead of two threads attempting to resize simultaneously, the atomic bool flag m_resizing is introduced, which was not included in [1]. Resize can only be called if a CAS call successfully flips the flag from false to true. Once resizing procedure is complete, m_resizing is set back to false via an atomic write.

The implementation of Set::Resize initializes all of the bins in the first HNode that have not already been initialized. A new HNode is created with a number of bins equal to double or half of the bins in the first HNode. The current first HNode is set as the pred pointer of the newly created HNode in preparation to make it a second HNode and the newly created HNode is set as the first HNode using a CAS call. If the CAS fails, it is only due to another Set::Resize call (triggerred by another Insert or Remove) successfully executing its resize. Since the HashTable has been resized even when the CAS fails, the CAS call does not need to be reattempted. This CAS call represents the linearization point of the resizing functionality of the HashTable. No bins are initialized in the first HNode immediately following a resize. Instead, the work for initializing the bins in the first HNode is distributed amongst the future calls that access those bins.

The Set::InitBucket method is used to populate a NULL bin in the first HNode. During this call, the NULL bin is populated with data from the second HNode, which is always initialized. The second HNode can either have half or double the size of the first HNode, denoting whether the last Set::Resize call grew or shrunk the hash table. If the table has been shrunk, then bin N in the first HNode inherits the values of bin N and N + numBins1 (where numBins refers to the number of bins in the first HNode). The proccess is slightly more convoluted if the table has grown. For such cases, bin N in the first HNode is populated with the entries in bin N % numBins2 that have a remainder of N when divided by numBins1. The bins in the second HNode are always frozen prior to being accessed. The Set::InitBucket method is linearizable since the bin pointers in the

HNode object are atomic and a CAS call is used to replace a NULL bin with a newly created one. That CAS also represents the linearization point of the initialization of a bin. The call can only fail if another call has came through and initialized that same bin, thus making it unnecessary to retry again.

The Set::Apply method represents a private helper function for processing an Insert or Remove. Apply can only be called on a bin in the first HNode. The designated bin is initialized if it happens to be NULL. An Insert or Remove should never fail due to a bin being frozen because only bins in the second HNode can be frozen. Each Insert or Remove call linearizes once it reaches the previously discussed linearization points within the underlying Set class. A size member (similar to the one in FSet) is implemented in the HashTable class, which is not part of the pseudocode in [1]. This variable is atomically incremeneted and decremented upon successful Insert and Remove operations. Again, this is only an estimate of the total number of nodes in the table and solely used as part of a resizing heuristic policy.

Set::Contains is the only other operation (besides InitBucket) that can access the bins within the second HNode, even if they happen to be frozen. Contains can collide with a Set::Union or Set::IntersectRemainder call, but that is acceptable because all of these functions only perform read operations on the Set class. Since Set::Contains does not have to initialize any bins, it is much more lightweight than a Insert or Remove. If Set::Contains had the ability to initialize bins, that would also significantly increase the contention on the InitBucket call since Contains is statistically more common than any other operation. The Contains call can complete in a prederminite number of steps even within the higher wrapping classes, thus maintaining the wait-free behavior that is introduced in the underlying Set class.

4 Concurrency

4.1 Progress Guarantees

Since our entire implementation is lock-free, at least one thread is guaranteed to make progress at any time. All finalized changes done to either the head node or its buckets are done using atomic CAS calls. If a thread fails to successfully update a value by failing the CAS call, it is only due to the fact that the value it expected was changed by another thread. In such cases, the other thread's CAS call must have succeeded. The concurrent behavior of each CAS call is explained in the Implementation section on a case-by-case basis. Generally, a successful CAS call represents the linearization of the parent function. Therefore, if another thread has linearized, it is making progress.

The Contains functionality of our implementation also provides wait-freedom, which is a stronger guarantee than lock-freedom. Each layer of a Contains call can execute in a predeterminant amount of steps, mostly depending on the number of Nodes that have to be traversed. Having a wait-free Contains that does not depend on CAS calls also helps decrease contention on shared data. The local InitBucket and Resize calls also provide wait-free progress because they do not require iterative CAS calls. As mentioned earlier, their CAS calls can only fail if another thread came through and already executed the same operation.

4.2 Correctness Conditions

All of the operations supported by our HashTable class and its subclasses are linearizable. Therefore, each operation has a single linearization point. Each linearization point is discussed on a case-by-case basis in the Implementation section. For most cases (with the exception of Contains), the linearization point is represented by an atomic CAS call. The thread locally processes a structure and utilizes the atomic CAS call to make that structure resident to all other threads. The order of all overlapping operations is determined by the point-in-time that each operation linearizes.

4.3 Synchronization Techniques

The synchronization techniques in this algorithm are composed of atomic variables and atomic calls, including atomic reads, writes, and CAS calls. Atomic calls are implemented to execute using a single operation that is provided as part of the hardware ISA. Therefore, such operations always execute entirely or not at all. Atomic reads and CAS calls are guaranteed to retrieve the latest version of a variable, which may not be in a local cache. Therefore, having multiple threads calling atomic operations on a variable can thrash the cache and lead to performance losses. Each atomic operation is discussed on a case-by-case basis throughout the Implementation section of this paper.

5 Profiling

All profiling data in the scope of this paper is profiled on a Intel i7-7820x based platform, unless "AMD" is included in the title of the data. The AMD chip used is an AMD Ryzen 7 1700. Both CPUs support up to 16 logical threads on 8 physical cores and are manufactured based on a 14 nm process. The Intel and AMD chip run at a base frequency of 3.60 GHz and 3.00 GHz respectively. In that sense, the Intel chip has a slight advantage.

Four data sets of 2000000 operations are used during the profiling runs. A large number of operations was chosen in order to reduce (1) variation introduced by the CPU frequency not being forced to a constant value and (2) overhead due to thread creation and destruction. All data sets are generated using the attached genHashTableData Python script. The four data sets vary based on (1) the range of values used as arguments and (2) the proportions of operation occurrences.

- InsRem-50-64K.dat
  - Proportions: 50% Inserts, 50% Removes, 0% Contains
  - Argument Range: 0-64,000
  - First 25% of operations are Inserts in order to pre-load the data structure

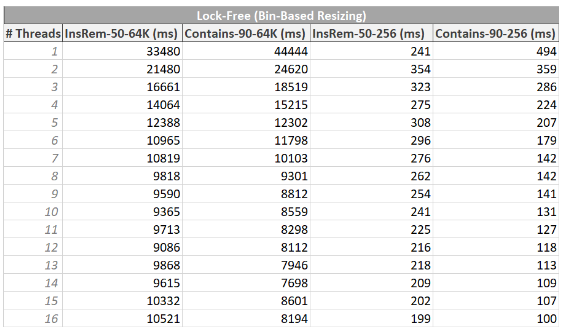
- Contains-90-64K.dat
  - Proportions: 9% Inserts, 1% Removes, 90% Contains
  - Argument Range: 0-64,000
  - First 5% of operations are Inserts in order to pre-load the data structure
- InsRem-50-256.dat
  - Proportions: 50% Inserts, 50% Removes, 0% Contains
  - Argument Range: 0-256
  - First 25% of operations are Inserts in order to pre-load the data structure
- Contains-50-256.dat
  - Proportions: 9% Inserts, 1% Removes, 90% Contains
  - Argument Range: 0-256
  - First 5% of operations are Inserts in order to pre-load the data structure

5.1 Resizing Heuristic Policy

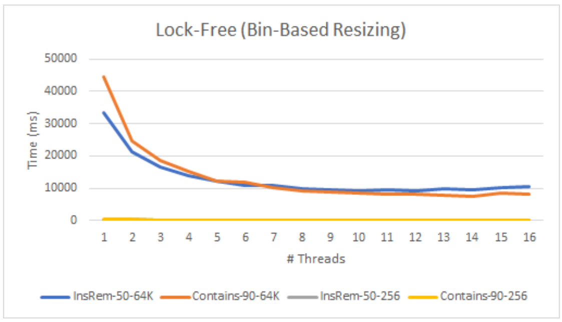
Determining when to resize and how often the resizing operation is done can significantly impact the performance and scalability of the algorithm. As a result, different heuristic policies are profiled in order to try to maximize the performance and scalability. The policies are based on the size and (upper/lower) threshold members found in the HashTable and FSet classes.

The original paper suggests to grow the number of bins when the bin being accessed becomes too large and to shrink whenever a random sample of bins are too small. A bin becomes too large when the number of values it contains exceeds the upper threshold value. Alternatively, a bin is too small when it has less values than the lower threshold value. Gathering a random sample of bins can get expensive due having to access multiple atomic values on a per-bin basis.

We explored a slight variation of the policy suggested by the authors. In our policy, downsizing is based on the size of the current bin being accessed, instead of a random set of bins. The data below shows the observed execution times when using this heuristic.

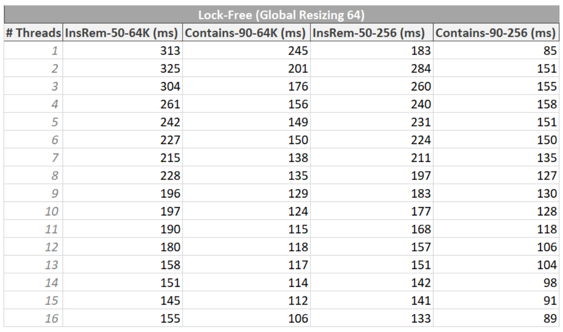


Lock-Free (Bin-Based Resizing)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	33480	44444	241	494
2	21480	24620	354	359
3	16661	18519	323	286
4	14064	15215	275	224
5	12388	12302	308	207
6	10965	11798	296	179
7	10819	10103	276	142
8	9818	9301	262	142
9	9590	8812	254	141
10	9365	8559	241	131
11	9713	8298	225	127
12	9086	8112	216	118
13	9868	7946	218	113
14	9615	7698	209	109
15	10332	8601	202	107
16	10521	8194	199	100

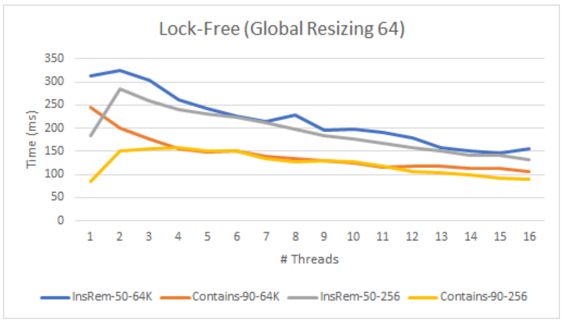


Reoccurring operation arguments are less likely in the datasets that have an argument range up to 64K. For such cases, the Insert and Remove operations are more likely to succeed and check the heuristic policy. The long execution times for these data sets are due to the resize method being called too often. Consecutive resize calls are being made because the size of the bins are not uniform.

We came up with an alternative heuristic that is agnostic to how uniform the sizes of the different bins are. This heuristic tracks all of the values in the hash table as a whole, instead of the size of the individual bins. This heuristic also introduces dynamic thresholds that do not require user input. The lower threshold is always equal to the number of bins in the hash table. This denotes that a single entry exists in each bin on average. The upper threshold is a multiple greater than two of the number of bins. Having a multiple greater than two is critical in order to avoid resizing in two consecutive operations. Various values for the multiples are profiled in the data included in the Appendix section. The data from the optimal multiple for our data set is shown below, which proved to be 64.



Lock-Free (Global Resizing 64)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	313	245	183	85
2	325	201	284	151
3	304	176	260	155
4	261	156	240	158
5	242	149	231	151
6	227	150	224	150
7	215	138	211	135
8	228	135	197	127
9	196	129	183	130
10	197	124	177	128
11	190	115	168	118
12	180	118	157	106
13	158	117	151	104
14	151	114	142	98
15	145	112	141	91
16	155	106	133	89



6 Transactional Implementation Using STM

Software Transactional Memory (STM) libraries allow atomicity to be implemented in software for a set of operations, known as a transaction. Each transaction must be entirely executed or not at all. Such behavior requires rewinding functionality that allows all the progress made by a transaction up to a certain point to be undone if another transaction has overwritten data shared between the two transactions. Rewinding a transaction can introduce additional overhead that is not present within other methodologies. Nevertheless, if a transaction is being rewound, it is due to another transaction making progress.

6.1 Sequential Hash Table

STM allows for a sequential implementation of a data structure to be made concurrent by defining its operations as transactions. In fact, the implementation must be sequential because the STL atomic operations fail to compile if they are defined within a GCC STM transaction. This section covers the implementation of the sequential hash table, which mimics the lock-free version but without the thread-safety functionality.

Two different implementations for the hash table bins are explored. One implementation (Sequential::Set class) utilizes a binary search tree for its quick logarithmic-time search functionality. The other implementation utilizes a linked-list (Sequential::List) for its ability to be easily split into two lists or combined with another list into a single one. The benefit of using the List class for the bins is only apparent if the hash table supports proper resizing hierarchy. Without proper resizing, the bins may grow too large and in which case the linear search time of the List make it inferior to the Set.

In addition to the traditional Insert, Remove, and Contains operations, the Set and List classes also include a Union and IntersectRemainder operations. Just like in the lock-free hash table, they are used when resizing the number of bins. Union combines two bins together while IntersectRemainder splits a bin into two based on the remainder value when dividing each element. These operations are simplified in the List implementation because:

- An existing list can be recycled
  - During a Union, the elements of the shorter List are inserted into the longer list.
- Since both Lists are always sorted, each of these operations require a single traversal through each List resulting in an O(m+n) complexity.
  - On the other hand, the time complexity of equivalent calls in Set is O(m * (n * log(n))) since each the tree has to be re-traversed for each Insert.

A set of verification tests are included in Tests.h that check the correctness of the Set and List classes. This code can be triggered by running the SequentialHashTable executable with an invalid set of parameters.

The InsRem-50-64K data set is used to compare the different variants of the Sequential::HashTable since it is the most stressful. The single-threaded results are shown below, which confirm the differences between the Set and List implementations.

- Sequential::Set is used for bin implementation and resizing is DISABLED.
  - 195ms
- Sequential::Set is used for bin implementation and resizing is ENABLED.
  - 118ms
- Sequential::List is used for bin implementation and resizing is DISABLED.
  - 6494ms
- Sequential::List is used for bin implementation and resizing is ENABLED.
  - 100ms

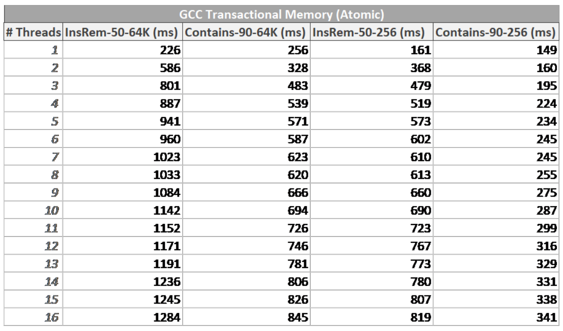
From here on, Sequential::List is used with resizing enabled. Various resize hierarchies are explored in this implementation, as for the lock-free hash table. Resizing based on the cumulative number of entries in the hash table again proves to be more efficient than resizing based on the size of individual bins. Resizing based on individual bins can lead to the number of bins constantly fluctuating if one bin has a lot of entries and another bin has very few. The lower threshold was chosen to be equal to numBins, since the optimal upper threshold showed to be 4 * numBins. Having thresholds that are ¿ 2 times apart eliminates the possibility of resizing twice in two consecutive operations.

- Resizing based on bin size. This can be enabled by changing the "#if 0" occurrences in Sequential::HashTable to "#if 1"
  - 170ms
- Resizing based on cumulative number of entries in hash table. Upper threshold is set to 2 * numBins. Lower threshold is set to numBins.
  - 123ms
- Resizing based on cumulative number of entries in hash table. Upper threshold is set to 4 * numBins. Lower threshold is set to numBins.
  - 100ms
- Resizing based on cumulative number of entries in hash table. Upper threshold is set to 8 * numBins. Lower threshold is set to numBins.
  - 109ms

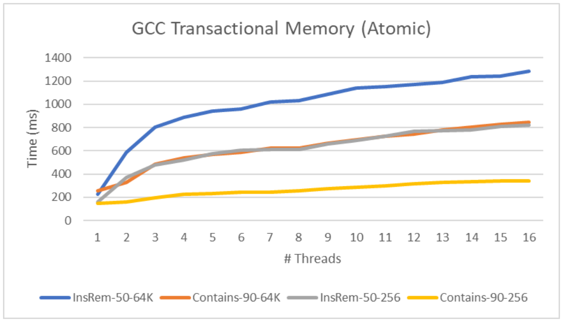
6.2 One Operation Per Transaction

The GCC transactional memory library is used in this implementation instead of RSTM because GCC is more commonly used and thus more likely to be streamlined. Also, the GCC transactions are implemented at the compiler level and are thus likely to be more efficient. Initially, each Sequential::HashTable operation is wrapped in its own transaction using the __transaction_atomic compiler directive. This compiler directive does not allow the compiler to insert locks, unlike the __transaction_relaxed directive [4]. SuperMalloc is used in this implementation as well in order to provide thread-safe memory allocations and deallocations. Introducing SuperMalloc did not create a noticeable change in performance.

The table below shows the results of having a single operation per each atomic transaction. The runtime of the single threaded test cases have likely increased due to the overhead of preparing and submitting the transactions. The runtime becomes progressively worse with each additional thread since there are various possibilities for collisions throughout every function. Contains is the only exception, hence why it shows the best performance.



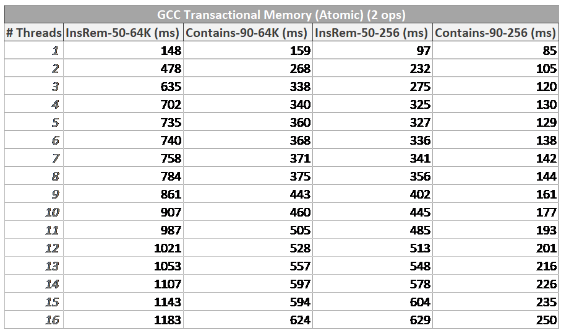
GCC Transactional Memory (Atomic)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	226	256	161	149
2	586	328	368	160
3	801	483	479	195
4	887	539	519	224
5	941	571	573	234
6	960	587	602	245
7	1023	623	610	245
8	1033	620	613	255
9	1084	666	660	275
10	1142	694	690	287
11	1152	726	723	299
12	1171	746	767	316
13	1191	781	773	329
14	1236	806	780	331
15	1245	826	807	338
16	1284	845	819	341



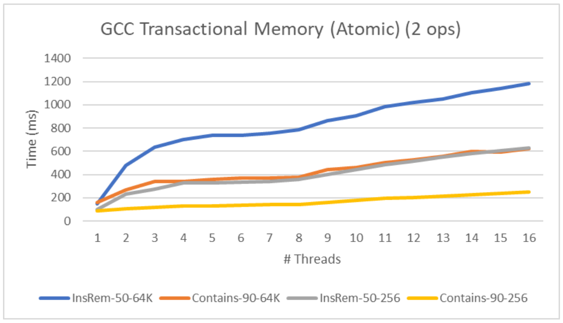
6.3 Multiple Operations Per Transaction

Packing multiple operations within a transaction proves to be more efficient for this kind of hash table. Having less transactions overall leads to less overhead dedicated to handling and submitting the transactions. This performance gain is directly reflected in the single-threaded case. However, having longer transactions leads to longer rewind procedures. This is likely why the rate of change drastically increases beyond 8 threads in the charts below, when collisions become much more likely.

The graph below shows the results from having two operations per transaction. This is significantly more performant than having a single operation per thread, since the gain of having to do bookkeeping for less transactions outweighs the loss of the transactions being longer. With more threads, this case is likely to become less performant than having a single operation per transaction.

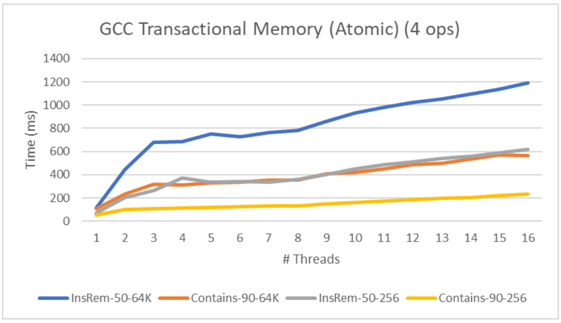


GCC Transactional Memory (Atomic) (2 ops)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	148	159	97	85
2	478	268	232	105
3	635	338	275	120
4	702	340	325	130
5	735	360	327	129
6	740	368	336	138
7	758	371	341	142
8	784	375	356	144
9	861	443	402	161
10	907	460	445	177
11	987	505	485	193
12	1021	528	513	201
13	1053	557	548	216
14	1107	597	578	226
15	1143	594	604	235
16	1183	624	629	250



The graph below shows the results from having four operations per transaction. It is more efficient than the either of two previous cases at lower number of threads. However, at higher number of threads it becomes no better than having two operations per transaction.

GCC Transactional Memory (Atomic) (4 ops)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	112	106	68	54
2	444	235	204	100
3	678	316	262	108
4	683	310	375	116
5	754	327	335	119
6	730	338	341	124
7	765	355	334	130
8	781	356	361	133
9	857	407	401	147
10	932	423	451	164
11	978	451	484	175
12	1021	487	508	187
13	1054	499	542	196
14	1093	533	558	204
15	1139	571	586	223
16	1189	563	621	234



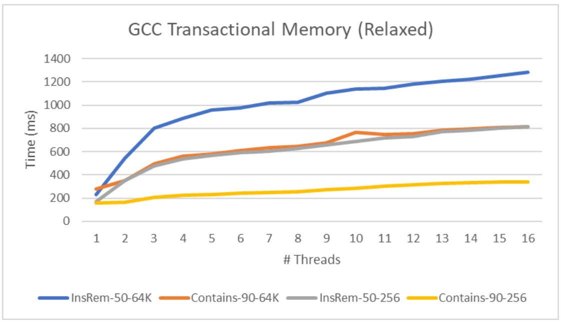
6.4 Improvement Attempts

This section covers several attempts to improve upon the STM implementation of the hash table. The only one that proves to be effective is to use coarse-grained locks instead of transactions for this kind of data structure.

6.4.1 Relaxed Transactions

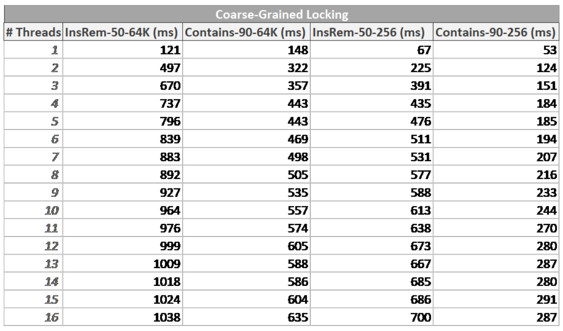
Having relaxed transaction instead of atomic ones allows GCC to determine if using locks for some cases can be more efficient than atomicity. Atomicity can become too convoluted to implement in some cases [4]. The data below show the results of using relaxed transactions for the case where a transaction includes only a single operation. It seems no different from the case where the transactions are strictly atomic. The results from having relaxed transactions for the cases where a transaction encompasses multiple operations are also like the ones with strictly atomic transactions.

GCC Transactional Memory (Relaxed)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	230	277	168	158
2	544	349	352	165
3	805	495	480	206
4	887	560	539	224
5	957	578	567	233
6	979	609	589	241
7	1019	632	602	250
8	1024	648	625	257
9	1102	678	660	271
10	1142	765	687	285
11	1147	750	717	305
12	1183	757	731	314
13	1206	784	773	325
14	1226	796	785	335
15	1256	806	800	340
16	1286	813	817	337

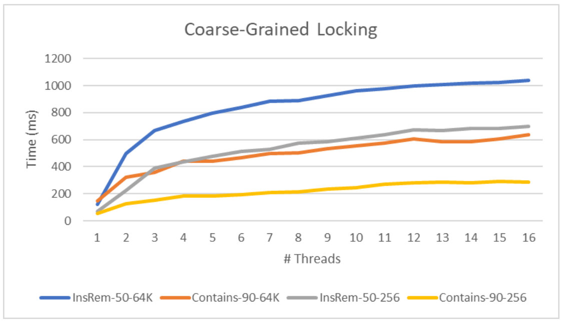


6.4.2 Replacing Transactions with Locks

Using locks (via std::mutex) instead of transactions showed approximately 20% higher performance. Mutexes synchronize the operations, but collisions are eliminated and thus rewinds are no longer necessary. The data below shows the results from wrapping each operation in a mutex. If multiple operations are wrapped, then the performance is likely to be even better since there will be less contention points.



Coarse-Grained Locking				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	121	148	67	53
2	497	322	225	124
3	670	357	391	151
4	737	443	435	184
5	796	443	476	185
6	839	469	511	194
7	883	498	531	207
8	892	505	577	216
9	927	535	588	233
10	964	557	613	244
11	976	574	638	270
12	999	605	673	280
13	1009	588	667	287
14	1018	586	685	280
15	1024	604	686	291
16	1038	635	700	287



6.4.3 Offsetting the Transactions

The synchronize keyword in GCC allows a block of code to always executed sequentially. Under the hood, this is likely implemented using locks and requires that the transactions be relaxed. Wrapping various points of the code to offset the transactions and reduce the number of collisions leads to an increased overall execution time. The collisions likely still occur, and additional overhead is introduced by serializing parts of the code.

6.5 Comparison to Lock-Free Implementation

The STM implementation of the Hash Table does not scale well with additional threads, in contrast to the lock-free implementation. The two implementations have comparable performance only when using two threads are being executed. However, the STM implementation is significantly simpler to apply to a sequential data-structure.

7 Conclusion

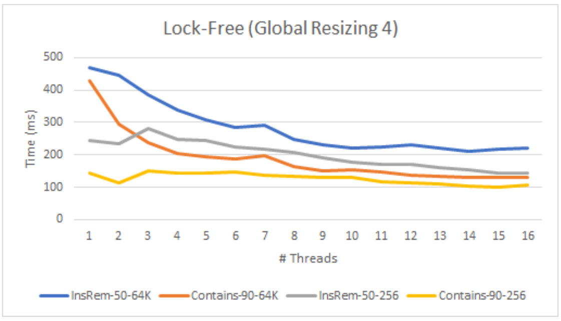
In this paper, we introduced our own implementation of the Lock-Free Dynamic-Sized Nonblocking Hash Tables created by Liu et. al in [1]. Our implementation differed from theirs in a few ways. Our Set object was a linked list as opposed to an array. The underlying Set impementation is not a core aspect of the Hash Table implementation and thus was not discussed in detail by the authors. We also added a resizing heuristic policy that was not part of the original pseudocode. However, we are assuming that the authors implemented their policy in a similar fashion. Multiple policies were explored and profiles, which are discussed in the Resizing Heuristic Policy section.

Re-implementing Liu et. al's data structure was not without its challenges. Converting their pseudocode to functional C++ code was often difficult, especially with certain operations that were not explicitly coding techniques, such as the union operation found in the pseudocode for the FSet object. Implementing the parts of the data structure not described by the authors was also troublesome. Coming up with our own heuristic policies to resize the hash table on took some time and we had to entirely change the Set from operating on an array to operating on a linked list, which involved creating a whole new class with all of its own methods.

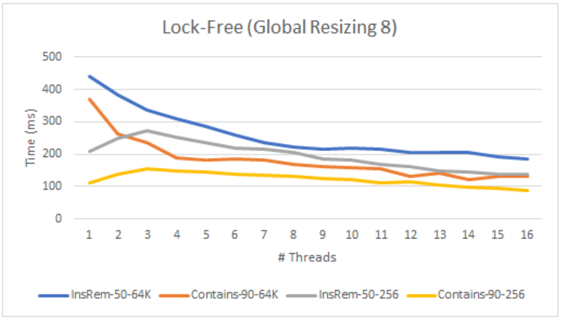
This data structure provides a lot of new advantages to existing alternatives. The biggest its dynamic property; being able to grow and shrink while previous hash table implementations were only able to grow. In an environment with limited memory this data structure is far optimal to its alternatives due to this key feature. It also allows keys to be moved among buckets during a resize operation which previous data structures have not done. Doing this during the resize allows this implementation to avoid sacrificing throughput or progress. Also, the buckets are unbounded and no assumption is made about the size of memory, which allows this data structure to be more flexible in its applications onto larger datasets. While the inner mechanisms of this data structure can be complicated, as there are four encapsulating layers of objects; once it has been successfully implemented the advantages of it far outweigh this.

8 Appendix

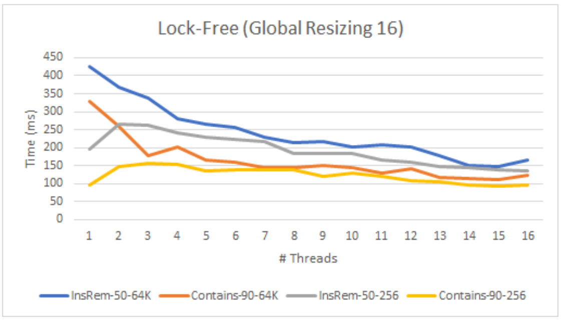
Lock-Free (Global Resizing 4)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	468	429	243	143
2	446	296	233	113
3	384	237	280	149
4	337	205	247	144
5	309	194	243	144
6	284	188	224	146
7	291	198	218	137
8	248	165	207	135
9	231	151	192	132
10	222	153	179	129
11	225	147	171	118
12	232	136	169	112
13	222	133	160	110
14	211	132	153	103
15	218	131	143	100
16	221	131	144	108



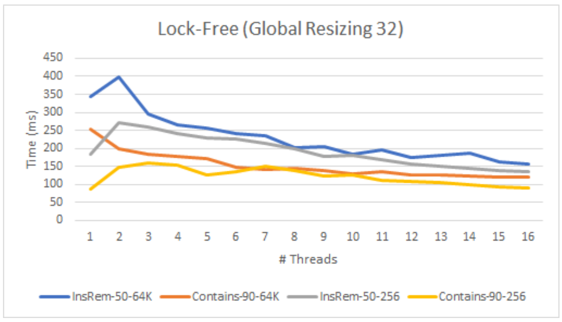
Lock-Free (Global Resizing 8)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	440	369	210	112
2	385	262	249	137
3	335	236	273	154
4	309	190	254	149
5	286	183	234	145
6	259	186	219	138
7	235	181	215	134
8	222	170	204	132
9	216	163	186	125
10	218	158	181	122
11	216	156	168	111
12	207	131	161	116
13	207	141	149	103
14	205	123	146	98
15	192	130	138	93
16	186	131	139	89



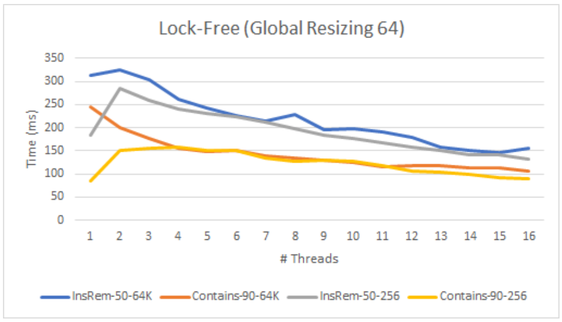
Lock-Free (Global Resizing 16)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	425	330	196	96
2	369	260	265	149
3	339	178	261	157
4	281	203	241	153
5	264	165	228	135
6	257	160	224	137
7	229	143	216	137
8	213	143	184	138
9	217	151	185	121
10	201	145	183	129
11	207	128	166	119
12	201	141	161	108
13	179	118	149	105
14	152	114	146	96
15	149	111	138	92
16	167	124	134	95



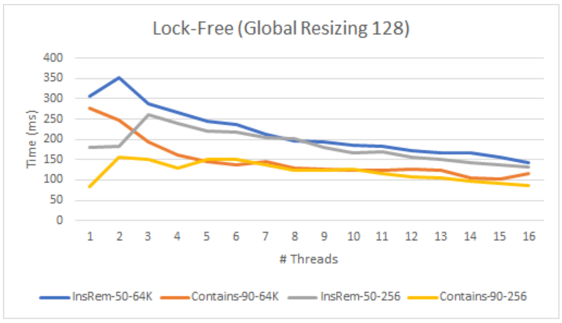
Lock-Free (Global Resizing 32)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	343	254	185	88
2	397	198	272	149
3	297	184	260	161
4	266	178	242	154
5	255	171	229	127
6	242	149	225	136
7	234	142	214	150
8	203	144	198	137
9	205	139	179	124
10	185	130	180	126
11	196	134	168	112
12	176	125	156	107
13	180	127	151	104
14	186	124	143	98
15	163	120	138	93
16	158	119	136	89



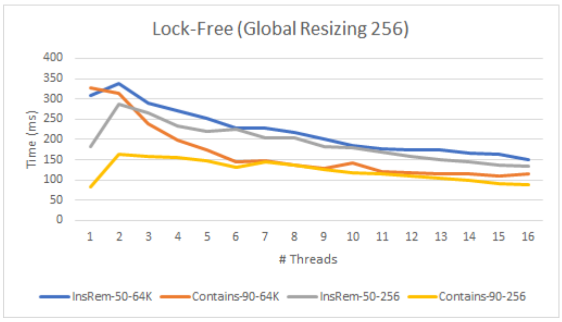
Lock-Free (Global Resizing 64)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	313	245	183	85
2	325	201	284	151
3	304	176	260	155
4	261	156	240	158
5	242	149	231	151
6	227	150	224	150
7	215	138	211	135
8	228	135	197	127
9	196	129	183	130
10	197	124	177	128
11	190	115	168	118
12	180	118	157	106
13	158	117	151	104
14	151	114	142	98
15	145	112	141	91
16	155	106	133	89

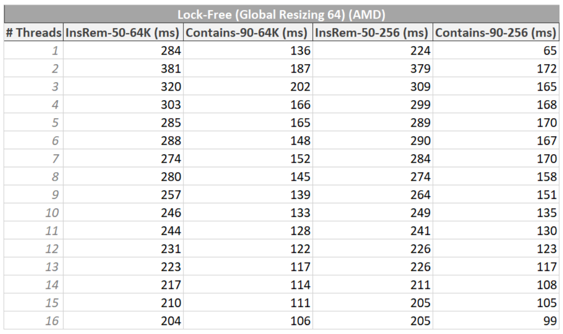


Lock-Free (Global Resizing 128)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	307	278	181	83
2	352	248	184	155
3	287	194	262	151
4	266	162	239	129
5	244	145	221	152
6	237	138	217	150
7	213	145	205	138
8	197	130	201	125
9	193	127	180	125
10	186	123	168	127
11	182	125	170	117
12	172	126	157	108
13	167	124	150	104
14	168	106	143	97
15	155	102	137	93
16	144	115	132	87

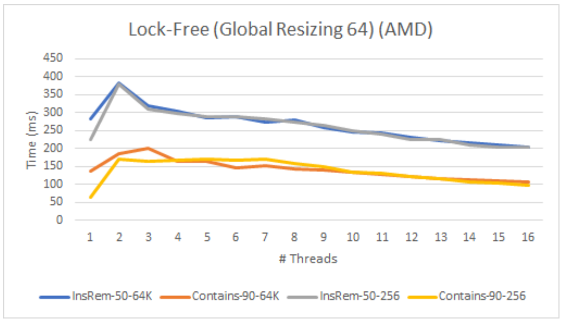


Lock-Free (Global Resizing 256)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	309	327	182	82
2	337	313	286	163
3	291	238	265	157
4	272	199	232	155
5	252	175	220	148
6	227	145	225	131
7	229	146	203	144
8	218	136	203	137
9	202	128	181	126
10	186	141	180	117
11	178	121	168	116
12	174	118	159	109
13	175	115	150	103
14	167	115	144	99
15	164	110	137	92
16	151	116	134	87





Lock-Free (Global Resizing 64) (AMD)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	284	136	224	65
2	381	187	379	172
3	320	202	309	165
4	303	166	299	168
5	285	165	289	170
6	288	148	290	167
7	274	152	284	170
8	280	145	274	158
9	257	139	264	151
10	246	133	249	135
11	244	128	241	130
12	231	122	226	123
13	223	117	226	117
14	217	114	211	108
15	210	111	205	105
16	204	106	205	99



References

[1] Y. Liu, K. Zhang, and M. Spear, “Dynamic-sized nonblocking hash tables,” *Proceedings of the 2014 ACM symposium on Principles of distributed computing - PODC 14*, p. 242–251, 2014.

[2] O. Shalev and N. Shavit, “Split-ordered lists,” *Journal of the ACM*, vol. 53, p. 379–405, Jan 2006.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier, 2012.

[4] M. Wong, H. Boehm, J. Gottschlich, V. Luchangco, P. McKenney, M. Michael, M. Moir, T. Riegel, M. Scott, T. Shpeisman, and et al., “Transactional memory support for c,” 2015.