Exploration of a Lock-Free Hash Table

Trevor Absher, Martin Dinkov

November 28, 2018

1 Abstract

This paper expands upon the algorithm presented in *Dynamic-Sized Nonblocking Hash Tables* by Liu et. al [1]. A specialization of their algorithm lies in "resizing in both directions: shrinking and growing." with the core of the algorithm being a "freezeable set abstraction, which greatly simplifies the task of moving elements among buckets during resize." [Liu et. al 2014]. Their algorithm is unique because previous attempts of a lock-free hash table (such as the variant by Shalev and Shavit [2]) did not support shrinking functionality. Furthermore, the freezable set abstraction in this paper expands on the normal operations of a set by adding a freeze method which prevents any thread from editing a given bucket while it is resized, simplifying the resize process [1].

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 [3].

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. Each method of the sequential hash table is treated as a seperate transaction. 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

In this section we will describe how we implemented each piece of Liu et. al's pseudocode while also implementing the code from Herlihy and Shavit.

3.1 Freezable Set

The Freezable Set object (FSet) was designed encapsulating the FSetOp object from the pseudocode and our own Set object. The FSet object contains only *set* and *ok* variables and an 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.

3.1.1 FSet Methods

We next encapsulated the methods that affect an FSet object into the FSet class itself in order to simplify the calls. The *HasMember* method takes in an integer and returns if it is found in the set or not. In our implementation, this calls the *Set.contains()* method. The *Invoke* method invokes either an add or remove operation based on the provided *FSetOp* object. It first checks to make sure this operation has not already been completed and that it is a valid operation. This method calls both the *Set.add()* and *Set.remove()* methods depending 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].

3.1.2 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] mention a set of integers, but do not go into detail as to how they implemented it. We decided on a Linked-List based Set class to represent the integer set of the FSet object. It is based off the Lock-Free version of the Set object given in

The Art of Multiprocessor Programming [3]. We made a slight modification to their Set object by adding a *size* variable, which represents the size of each bucket. This variable is necessary as this is how our heuristic policies dictate when the buckets need to be resized. Initially, we made the size of the buckets 1000, with an estimate of 500,000 operations per test case. This Set class is made up of Node objects, also found in [3]. The only other change we make to the code provided by Herlihy and Shavit [3] is removing the *key* variable from the Node class. Since we are using integers for the values we can simply sort the table on them, so we do not need a hash key. The Window object from [3] is also included for usage in traversing the list.

3.2.1 Set Methods

All methods of our Set object are the same as those provided by [3], with an additional method *Union*. A union operation is specified in the *Invoke* method of [1] but in order to get the proper functionality we needed to abstract it into a method. It takes in a Set object and merges it with the Set object that called the method, returning the Set with all the data from both.

3.3 Hash Table

Next, we will go into details of how we implemented the Hash Node (HNode) object from [1]. We created a HashTable class with an HNode struct and encapsulated all the methods that operate on HNodes into the HashTable class.

3.3.1 Hash Table Methods

The Hash Table has three public methods: *Insert*, *Remove*, and *Contains*, all specified in [1]. There were no differences in our methods other than in *Contains* where we made sure to atomically load the head node and the exact specifications of the heuristic-policy in *Insert* and *Remove*. Since [1] did not specify a heuristic policy there was no way we could implement theirs; in addition to us often changing the policy so we could experiment with which one performed the best.

The Hash Table also has three private methods and one private variable. The *m_head* variable is a pointer to an HNode object that is currently the head of the Hash Table. This variable is atomic, and all operations will be performed on the HNode that is the head of the Hash Table. Again, the only difference between our implementations was that we began by atomically loading the head node, which is never explicitly stated in the pseudocode provided by [1].

The *Resize* method operates exactly as it does in the provided pseudocode from [1]. The Compare and Set call is done by making the head node atomic and then calling *compare_exchange_strong*. The *InitBucket* method also operates the same as in [1] with only a few differences to point out. The Set object initialized in this method is of our Set class implemented with the pseudocode from [3] and the Union performed on line 48 of the Lock-Free method of [1] is done using the *Union* method of Set. Finally, the

Compare and Set is done using a call to *compare_exchange_strong* to put it into the HNode atomically since the buckets are also atomic.

3.3.2 HNode

The HNode struct contains an array of atomic buckets in the form of FSet objects; denoted as *buckets*. In order to determine which bucket a value goes into, we use the simple formula specified in [1]: index = k **mod** size. The *size* variable denotes how many buckets there currently are in the given HNode. The *pred* variable is of type HNode and points to the previous HNode in the Hash Table. This HNode will be either half or twice the size of the current HNode, depending on what resize operation was called to change it.

4 Concurrency

In this section we will discuss how the algorithm we implemented meets concurrency requirements.

4.1 Progress Guarantees

Our algorithm does not use any locks. All finalized changes done to either the head node or its buckets are done using the *compare_exchange_strong* method, which is a version of the well-known Compare and Set method. If a thread fails to successfully update a value due to failing this *compare_exchange_strong* call, it is likely due to the fact that the value it expected to be there was not, and thus the thread did not update the value to what it intended. This would only occur if the value had changed due to another thread coming in slightly before the first thread and changing the value. However, this means the second thread did successfully update the value. The first thread will try again to update the value (due to the *resp* variable) but it may never succeed due to the same problem. If this same problem is occurring over and over, we know that the other threads are succesfully completing their operations. Since at least one thread can be guaranteed to successfully complete their operations and update the value, our algorithm is lock-free and thus is guaranteed to make progress. Even if a thread crashes during the *Freeze* method and leaves the Set immutable without properly creating a new head node, the next FSetOp to attempt to insert or remove a value will fix this. The insert or remove will fail but the check for the heuristic policy will still trigger and this thread will then resize the Hash Table.

4.2 Correctness Conditions

All changes performed by inserts and removes will be performed only on the head node, and only on buckets. Both the head node and its buckets are atomic variables, which are guaranteed to only be edited by one thread at a time, and all threads will read the same value when reading an atomic variable. Since all updates to the head node and its

buckets are done using *compare_exchange_strong*, we know that no threads will overwrite each other. The official updates to head and its buckets linearize at this method call, while the minor updates such as the insert, remove, and contains calls linearize when the *op.done* variable is set to true. All the values for each slot in the sets of the buckets are also loaded atomically, so we know that we will receive the correct value when it is loaded. The *Freeze* method also guarantees that no inserts or removes are lost during the time it takes a thread to successfully resize the Hash Table.

4.3 Synchronization Techniques

The main synchronization techniques used in this algorithm is the use of atomic variables and the *compare_exchange_strong* method. The atomic variables ensure that every read to these variables will be synchronized so that every thread sees the same value at the same time. The *compare_exchange_strong* method allows us to ensure only one thread will update the head node or its buckets at a time, protecting us from values getting incorrectly overwritten or certain calls returning invalid responses. However, the key synchronization technique in this algorithm is the *Freeze* method. The *ok* bit getting set to false stops every thread from editing the data of that HNode until the resize is complete. While this does interrupt progress for a time, this ensures that no data is lost while we switch the *m_head* variable to a new HNode. If *Freeze* were not called, every insert and remove performed after the first line of *Resize*, where we atomically load the current value of the head, would be lost.

4.4 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.

4.4.1 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.

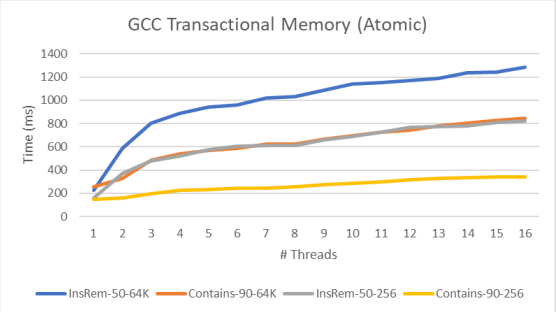
- Resizing based on bin size. This can be enabled by changing the "#if 0" occurrences in Sequential::HashTable to "#if 1"
  - 170ms
- Resizing based on cumulative number of entries in hash table. Upper threshold is set to 2 * numBins. Lower threshold is set to numBins.
  - 123ms
- Resizing based on cumulative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

4.4.2 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

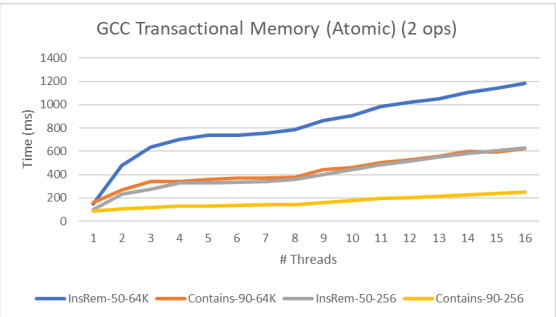


4.4.3 Multiple Operations Per Transaction

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.

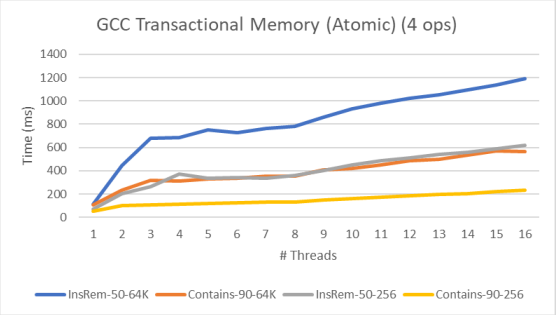
The graph below shows the results from having two operations per transaction. This is significantly more performant than having a single operation per thread, since the gain of having to do bookkeeping for less transactions outweighs the loss of the transactions being longer. With more threads, this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



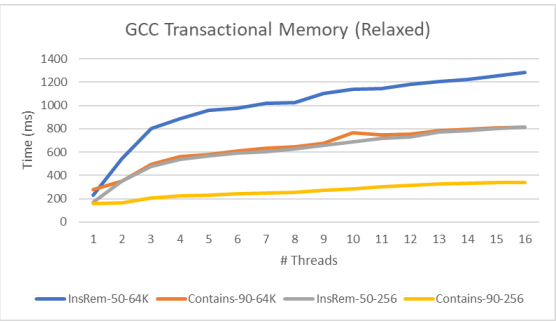
4.4.4 Improvement Attempts

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.

4.4.5 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.

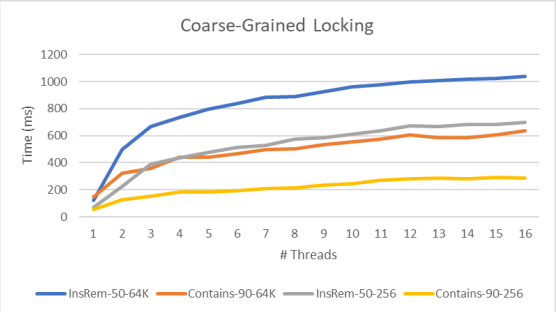
GCC Transactional Memory (Relaxed)				
# Threads	InsRem-50-64K (ms)	Contains-90-64K (ms)	InsRem-50-256 (ms)	Contains-90-256 (ms)
1	230	277	168	158
2	544	349	352	165
3	805	495	480	206
4	887	560	539	224
5	957	578	567	233
6	979	609	589	241
7	1019	632	602	250
8	1024	648	625	257
9	1102	678	660	271
10	1142	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



4.4.6 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



4.4.7 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.

4.4.8 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.

5 Performance Evaluation

In this section we will compare the performance of our implementation of the algorithm versus the official algorithm provided by Liu et. al in [1]. We also will compare the performance of our different heuristic policies. Finally, we measure the performance differences between our initial implementation in a concurrent data structure and our final implementation where we converted it to a transactional data structure.

6 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. First, our Set object was a linked list as opposed to an array. While the array has superior locality of memory, the authors of [1] did not provide information as to how they implemented it. Next, we implemented a size variable and a few threshold variables. While these were not explicitly stated by the authors of [1], it is more than likely they had something similar in place otherwise they would not have been able to have anything to determine their heuristic policy off of. Our size variable tracked the

current size of each bucket, while the lower threshold was the bound for shrinking the Hash Table, and the upper threshold was the bound for growing the Hash Table. The max size variable provided a maximum number of buckets so that the memory would not grow exponentially if threads continually needed to make buckets by adding similarly hashed values. Our final difference was our heuristic policy. Again, this difference was due to the authors of [1] not stating exactly what heuristic policy they used, as this was not the goal of their paper. We first implemented a simple policy, growing when the current bucket was too big, and shrinking when it was too small. Next, we tried what the authors suggested and grew when the current bucket was too big and shrunk when a random sample of buckets were all too small. Finally, the last policy we tried (which also performed the best) was to resize based on the cumulative size of all the entries in all buckets.

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. Using the size variable of each bucket also proved challenging. Without using a Descriptor object, the size variable was not perfectly thread safe. Retrieving the size variable at any given time may not have been the true correct value of the size by the time it was used. However, implementing a Descriptor object would have complicated the data structure even more, so this was determined to be acceptable. To make up for this, the maximum size of each bucket was not enforced, but a suggestion. The hash table would be resized when a bucket grew close to its maximum size, but if threads attempted to add values to it before this size was properly read by the resize operation, the values would still be added to the bucket and would not be lost.

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.

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.