# An implementation of Non-blocking Unordered Lists

Abhianshu Singla
University of Central Florida
Orlando, Florida
abhianshu@knights.ucf.edu

## ABSTRACT

This paper provides a detailed description of the unordered linked list algorithms along with their key synchronization techniques, progress guarantees, and correctness conditions. The implemented algorithms are the lock-free list, wait-free list, modified lock-free list, coarse-grained list, and a transactional list using Rochester Software Transaction Memory (RSTM). Their performance is evaluated based on the the runtime each algorithm takes with a different distribution of operations and different transaction sizes.

## 1 INTRODUCTION

Among all the non-primitive Data Structures, Linked List is the most popular and widely used data structure. It can be used to construct various other abstract data structures such as Stacks, Queues, Associative Arrays-Maps, and Dictionary, or it can also be used on its own. Due to its various applications, a lot of researchers have put their efforts into making it a concurrent object. The very first and basic approach is to use coarse-grained locks in which a thread acquires the lock at the start of the method call and releases it before returning the call. This makes the list fundamentally sequential and it performs worse than the sequential list in high contentions as every thread will try to acquire the lock, but only one will succeed. Another approach is to use fine-grained locks in which the locks are applied to individual nodes with hand-over-hand coupling. It evidently performs better than coarse-grained locks, but not by a significantly high amount. Like the coarse-grained list, it is also a blocking list as the thread which has the lock at first node is blocking all the other threads which are waiting to acquire locks. This approach is also not suitable for high contentions/loads. So, a different approach (optimistic synchronization technique) is taken in which a thread tries to acquire the lock only when the corresponding node is located in the list and if it fails to acquire the lock, it traverses the list again. However, in most of the scenarios, the cost of traversing the list again is much more expensive than the lock-based list when the validation check fails at a seemingly promising node.

In practice, only 9% of all the operations are *add* and 1% are *remove*, while the remaining are *contains* method. This implies that making the *contains* method lock-free can empirically improve the performance. This could be attained with Lazy synchronization where the main idea is that the nodes are first logically deleted and then physically deleted from the list. The *add* and *remove* method physically delete all the intermediate marked nodes while traversing the list, but the *contains* method ignores the intermediate marked nodes and returns true or false based only on the found node and its mark. It still has a blocking *add* and *remove* method. Generally, it was believed that non-blocking algorithms are better than the blocking ones. In non-blocking synchronizations, all the method calls are wait-free. A simple way to achieve it is by updating the next pointer and the mark update in a single atomic operation using Double-Compare-And-Swap Instruction. Unfortunately, a Double-Compare-And-Swap operation is much more expensive than a single Compare-And-Swap operation.

The first few lock-free algorithms for Concurrent List were designed for only Ordered Lists where the keys are completely ordered. Herlihy [2] proved that all the concurrent objects can be universally constructed as a wait-free structure, using the consensus protocol in a hierarchical manner such that the object at a higher level cannot be constructed as a wait-free object through the objects at lower levels. Following this proof, a wait-free Linked List using the universal fast-path-slow-path methodology was implemented which turned out to be less efficient than expected due to high overloads. In this paper, a lock-free algorithm based on the Lazy Search Phase is implemented in which a node is first inserted at the start of the list and then the thread searches the list to complete its operation. To further make it wait-free, a descriptor object is used to hold the information of each thread and each thread makes an announcement of its intention so that the other threads can look at it and help them succeed. The idea for making it wait-free is taken from wait-free Queue implementation by Kogan and Petrank [3].

In this paper, Section 2 discusses the related work and Sections 3 and 4 describes the lock-free and the wait-free list algorithms respectively. In Section 5, a brief introduction of Software Transaction Memory List is given and in Section 6, the effectiveness and performance evaluation of both the Lock-free and Wait-free List is presented along with the STM List, Coarse-grained List, and a Modified Lock-free List.

## 2 RELATED WORK

The introduction of universal lock-free data structures provided a basic approach for implementing concurrent data structures, however, it incurs a huge amount of overhead. The performance can be enhanced by reducing the number of accesses to a shared memory. Valois [8] provided the first non-blocking list algorithm which uses only the CompareAndSwap instruction. His implementation used auxiliary cells between every two standard nodes which is similar to having a linked list of size double than the normal size. Greenwald [1], in his Ph.D. dissertation suggested a Double-Compare-And-Swap for updating two different memory locations in a single atomic instruction. This instruction makes the concurrent Linked List simple to execute and understand. Unfortunately, many modern multi-processors don't support Double-Compare-And-Swap instruction.

Harris [4] provided a non-blocking algorithm using a marking technique in which the lower bit of the next pointer is marked to signify the logically deleted node in the list. This is based on the concept of Lazy Synchronization in which the nodes are first marked

and then physically deleted by itself or other helping threads. After this, Heller [6] implemented wait-free contains operation with lock-free insert and delete operations. Kogan and Petrank [3] designed a wait-free queue with a priority-based helping scheme in which the faster threads helps complete the slower threads operations which is based on the fast-path-slow-path methodology. Their implementation was developed on the Michael and Scott [5] lock-free queues.

## 3 LOCK-FREE LIST

The lock-free list algorithm acts as the building block of the wait-free list algorithm. Similar to a standard Linked List, it supports all the three basic operations - *insert*, *remove* and *contains*. The *insert* method adds the node to the list if it is absent from the list and returns true, otherwise, it returns false. Similarly, the *remove* method deletes the node from the list if it is present in the list and returns true, else it returns false. The *contains* method returns true only if the node is present in the list and returns false if it is not present. The add and remove operations modify the list, while contains only scans the list.

Apart from the standard List node which has a *key* and a *next* pointer, it also has a State value which helps in making the operation split into two phases- lazy search phase and physical insertion/deletion. The *key* is the item's value and the *next* value contains the reference to the next node in the list. A node's state will be one of the four values - *INS*, *DEL*, *DAT*, or *INV*. If a node's state is *INS*, it implies that a thread is trying to insert that node and if a node's state is *DEL*, then a thread is trying to remove the node. The node is logically present in the list if its state is *DAT*. A *INV* state implies that the node is logically absent from the list. It also has a sentinel head which points to the first element in the list.

### 3.1 Algorithm Design

In this Lock-Free Unordered List implementation, an intermediate node is inserted at the start of the list and then the thread scans the list and completes the operation depending on the search result. So, there are two phases for every *insert* and *remove* operation. In first phase, the new node is enlisted to the *head* using *enlist* method and in second phase, actual operation is done using *helpInsert*/ *helpRemove* method.

In the *insert* method, a new node $h$ with a state $INS$ is being inserted at the head using atomic *CompareAndSwap* Instruction. As *head* is a shared variable, many other threads might be adding the intermediate node to *head* at the same time, so a proper synchronization is required at this point. After successful insertion of the intermediate node, it starts scanning the list and deletes the nodes with $INV$(invalid) state until it founds a node with same *key* value or reaches the end of the list. If the detected node has state $DAT$ or $INS$, it means the node is already in the list, hence change the state of $h$ to $INV$. However, if the detected node has state $REM$, then some other thread is removing it from the list which logically means that the node is not present in the list. The same case happens when it reaches the end of the list. In this case, change the state of $h$ to $DAT$. The insert method tries to change the state of $h$ only one time using *CAS* instruction. The *CAS* fails only if the state of $h$ is changed from $INS$ to $REM$ by a thread with *remove* operation on the same key and other threads will only change the state to $INV$ when the prior state is $DAT$. The failure of *CAS* implies that some other thread is trying to remove this node, so make a Remove method call from this node and change its state to $INV$. Figure 1 and Figure 2 shows the two examples of insert operations.

The *remove* method's initial phase of adding the node at the head is similar to the *insert* method except that the state of the newly created node is *REM* now. In the scanning phase, it also physically removes all the passing nodes with state $INV$. If it founds the same key value node with a state *REM* or reached the end of the list, then the key is not present in the list, mark $h$ state to $INV$ and return false. However, if the found state is $INS$, then try to change it to *REM* using *CAS*. At this stage, either another *remove* call changes the state to *REM* before this *CAS* invocation or another *insert* method changes its state to $DAT$. If it is changed to *REM*, then its work is done and if it is changed to $DAT$, then change it to $INV$, mark $h$ state to $INV$ and return true. The contains method returns true if the node is physically in the list i.e. state is $DAT$ or the node is logically present in the list i.e., the state is $INS$ and returns false otherwise. Figure 3 and 4 shows the two examples of the *remove* operation.

### 3.2 Progress and Correctness

For synchronizing the concurrent threads, only the atomic CompareAndSwap Instruction is used. Also, Atomic *CompareAndSwap* requires its first argument to be atomic, therefore *head* and *state* are also atomic. The *contains* operation just scans the list, so it doesn't use *CompareAndSwap* instruction. The *insert* operation uses two *CompareAndSwap* instruction and *remove* operation uses only one *CompareAndSwap* instruction if the *state* is not $INS$ and two *CompareAndSwap* if the *state* is $INS$.

As there is no CAS instruction in contains operation which implies that it doesn't share any race condition with *insert* or *remove* operations. Hence, *contains* is always completed in a fixed amount of time and is a wait-free operation. However, *insert* and *remove* operations are not wait-free as *enlist* method is not wait-free, although *helpInsert* and *helpRemove* methods are wait-free. The *enlist* method is not wait-free because the threads can fail each time they try to insert the newly created node at the head. Still, this failure at *CAS* indicates that some other thread has successfully changed the head, therefore, it is lock-free. The *helpInsert* and *helpRemove* methods are wait-free as the thread will return from it either if it reaches the end of the list or it founds the node.

The linearization points for both insert and remove is at successful CAS updating the *head* to the newly created node. If the node is successfully inserted at the head, then all the concurrent threads racing for *head* will see its effect as the key is logically present(or absent) in the list if the state is INS(or REM). *contains* linearizes when the corresponding key's state is stored to a variable and if the list is empty, it linearizes at the assignment of the current node to head.

### 3.3 Modification

In the above implementation, *contains* operation just scans the list and returns true or false based on the state of the node. However, while traversing the list, it doesn't delete the nodes with $INV$ states

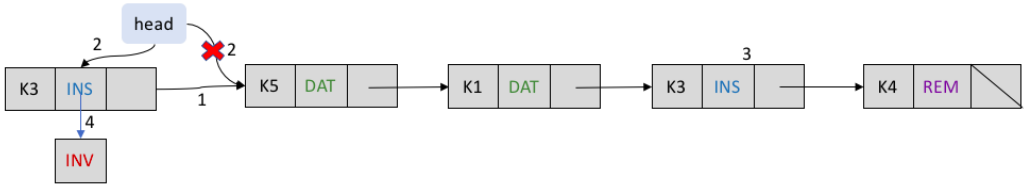


**Figure 1: Insert(K3) Operation**

Thread creates the node with Key K3 and state INS and constantly try to push it at the head of the list using CAS till it is successful. After finding K3 with state INS, changes the created node state to INV as K3 is logically present in the list.

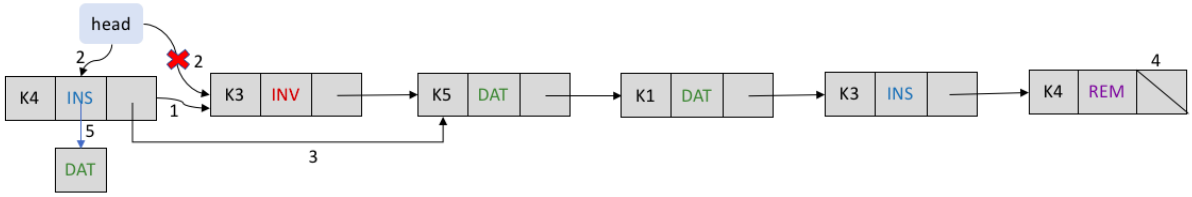


**Figure 2: Insert(K4) Operation**

Thread deletes node K3 as the state is INV while traversing the list and after an unsuccessful search, it changes the state of node K3 to DAT.

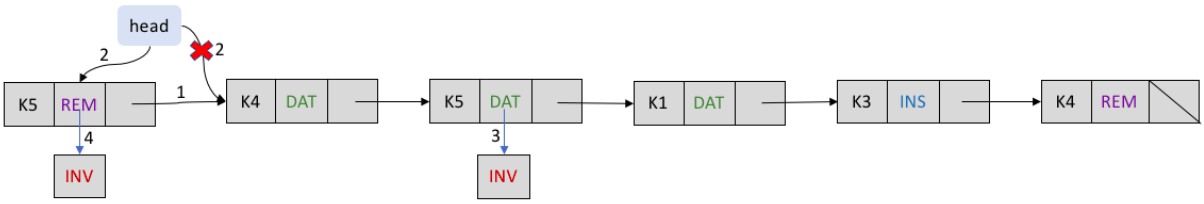


**Figure 3: Delete(K5) Operation**

Thread founds the node K5 in state DAT, so changes its state to INV (logical deletion) and also changes inserted node K5 state to INV as the operation is completed.

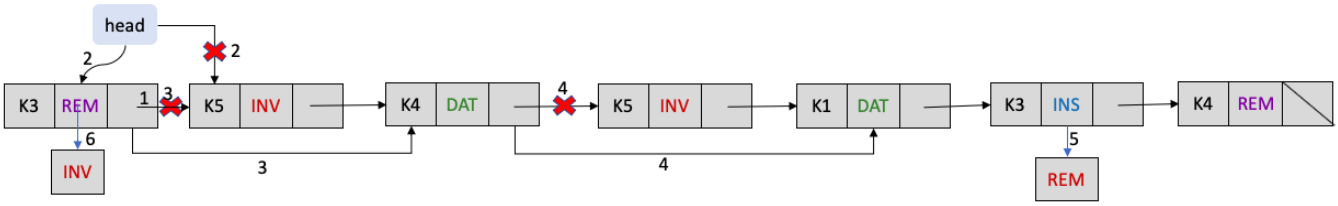


**Figure 4: Delete(K3) Operation**

Thread deletes all the nodes in state INV, finds K3 in state INS, changes it to REM and also changes the new node having key K3's state to INV. The state is changed from INS to REM as the thread which invoked insert(K3) has not finished yet, so it will eventually fail at CAS and will call remove(k3).

and moreover, this deletion of *INV* state nodes doesn't require a *CompareAndSwap* instruction. Also, in practical scenario, there are 90% *contains* method. If the list is large enough because there are many nodes with state *INV*, then every *contains* operation will have to go through a large list size. However, if the *contains* operation also deletes these node, then it is helping other *contains* operation by physically decreasing the list size. The *insert* and *remove* operations works similar to the operations described above.

## 4 WAIT-FREE LIST

A wait-free list is implemented by developing on the previous lock-free list by making the enlist operation wait-free as all other operations are already wait-free. If each thread can make its state known to other threads, then the later threads can help finish the operation of the previous threads. To achieve this, each thread has a descriptor object which consists of a phase id, a pending Boolean field and a pointer to the enqueueing node. The phase id is given by a counter which increments every time a thread creates a new descriptor object. The pending field tells whether the insertion of an enqueueing node to head has been completed or not. The pointer

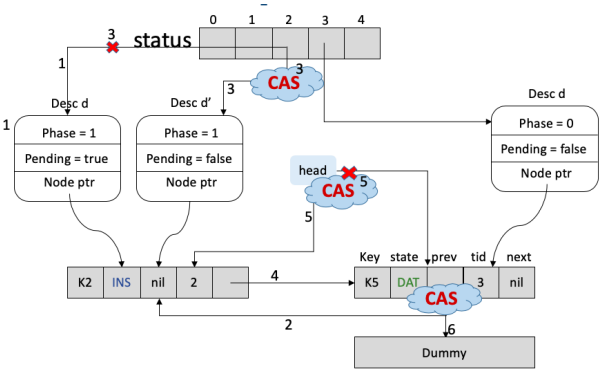


**Figure 5: Wait-free Insertion of a node at head of list**

to all the descriptors are stored in an array status with index equals to thread id. The node also has a *prev* pointer and a $thread - id$ in addition to the fields - key, *next* and *state*.

## 4.1 Algorithm Design

The basic idea behind the wait-free implementation is to let different enlist methods help each other to complete and also ensure that it completes its own operation in bounded number of steps. To achieve this, each enlist method announce its operation before starting the operation and each threads announcement can be checked from the status array which contains the reference to every threads descriptor. Figure 5 shows the steps of enlisting the new node to the head.

Whenever a thread invokes a insert or a delete method, it announce its operation by creating a new descriptor $d$ with pending state false and a pointer pointing to the newly created node $h$ and storing $d$ in status array. All the subsequent steps can be performed either by the thread itself or the helping threads with a higher phase number than this thread's phase. Accordingly, this thread will help only those threads that arrive before it. The thread finds the node pointed by *head* and then changes its *prev* value to point to node $h$ using *CAS* instruction as other threads are trying to do the same thing. After $h$ is linked to the *prev* of the first node in the list, the thread then creates a new descriptor with *pending* set to false and updates the descriptor in the status array so that the helping threads can see its updated status. This update is also achieved using *CAS* instruction so as to prevent the concurrent helpers from retrying. Then the thread updates the new node's *next* pointer to point to the node previously pointed by head. Finally, it updates *head* to point to the new node $h$ and clears the *prev* field of the node previously pointed by head.

## 4.2 Progress and Correctness

In wait-free list algorithm, *enlist* operation uses three *CAS* instructions for enlisting the new node at the head of the list as opposed to one *CAS* instruction for the lock-free *enlist* method. Now, the *insert* operation uses four *CAS* instruction and *remove* operation uses three *CAS* instruction if the *state* is not *INS* and four *CAS* if the *state* is *INS*. Apart from the atomic fields *head* and *state*, *Prev* and reference to descriptor objects are also atomic.

The *contains*, *helpInsert* and *helpRemove* operations are similar to the lock-free list and all three methods are wait-free. Also, the *enlist* method is modified to make it wait-free as the threads help each other and also ensure to complete its own execution in a bounded number of steps. Now, *insert* and *remove* operations are also wait-free.

The *insert* and *remove* methods linearize when the newly created node's *next* field is set to the node originally pointed by the *head* and the *contains* method linearize when the corresponding key's state is stored to a variable and if the list is empty, it linearizes at the assignment of the current node to head.

# 5 STM TRANSACTIONAL LIST

The traditional method of using locks to provide synchronization to different threads is prone to deadlock, starvation, priority inversion or convoying. Deadlocks can occur if threads try to lock the same objects in different order. Starvation can occur if a thread is continuously denied access to lock the object. Priority inversion occurs when a high-priority thread interrupts a low-priority thread which has the lock needed by the high-priority thread. To avoid the problems of locking, atomic primitives are used such as *AtomicInteger*, *orCompareAndSet* to update a single memory word. These primitives also have a high overhead associated with them. Also, if we want to update two different words such as head and next pointer of the node, then we need two *CompareAndSwap* instructions. These instructions are atomic individually, but not atomic if we want the combined effect. So, a *MultiCompareAndSet* is used to update multiple words atomically, i.e., if a check on any single word fails, then the instruction will return false. Both the locking and the atomic primitives have a serious drawback that they cannot be composed easily. It is possible to compose them, but it can be very difficult to implement sometimes resulting in complex algorithms.

Transaction Memory promises ease of code with great performance and no worries about deadlock, starvation or any of the problems associated with locking. It checks for the corner cases under the hood. It also provides serializability which is composable as opposed to linearizability which is not. Serializability defines atomicity for entire transactions whereas Linearizability defines atomicity of individual objects. As of now, there is no support in hardware for the transactional memory. So, Software Transactional Memory is used to control access to the shared memory in concurrent programming. A Memory Transaction is a collection of reads and writes executed atomically. There are various STM libraries and in this project, I used Rochester Software Transactional Memory (RSTM). RSTM provides the following features - It uses only one level of indirection to access data objects which helps in the reduction of cache misses. It avoids dynamic allocation of per-object or per-transaction metadata. It avoids tracing or reference counting garbage collection. It supports a variety of options for conflict detection and contention management [9].

To begin with the STM List, I implemented a sequential list first. Then I applied *TM_READ* instruction to read the shared memory, *TM_WRITE* to write to a shared memory, *TM_ALLOC* to allocate the memory to a shared object, and *TM_FREE* to free the shared object in a transaction. With the given API and using rstmlist.hpp as

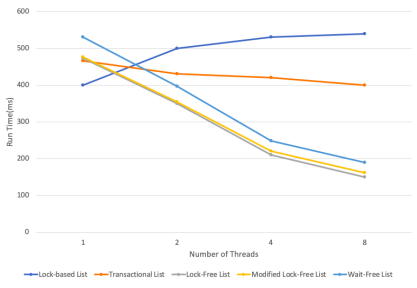


**Figure 6**
T_Size = 1, n_Transactions = 10000
9% insert 90% contains 1% remove

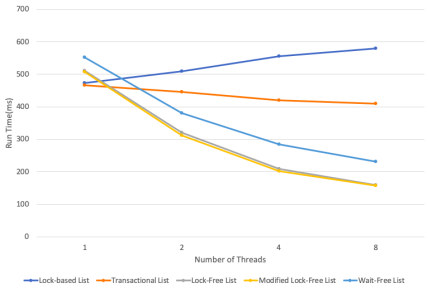


**Figure 7**
T_Size = 1, n_Transactions = 10000
20% insert 75% contains 5% remove

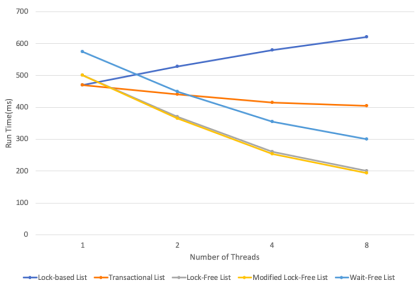


**Figure 8**
T_Size = 1, n_Transactions = 10000
30% insert 60% contains 10% remove

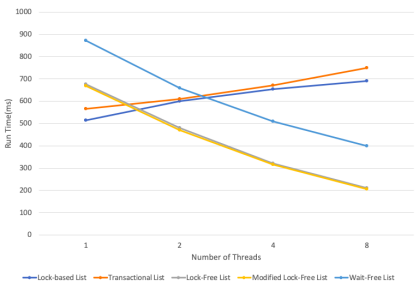


**Figure 9**
T_Size = 3, n_Transactions = 500000
33% insert 34% contains 33% remove

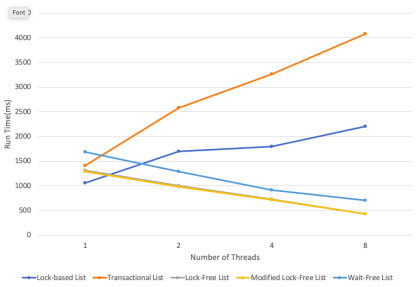


**Figure 10**
T_Size = 6, n_Transactions = 500000
33% insert 34% contains 33% remove

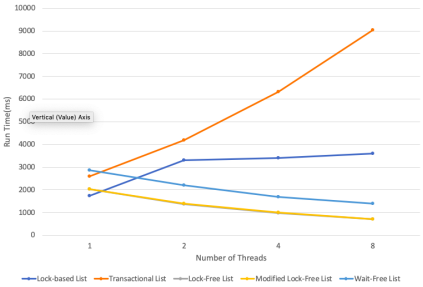


**Figure 11**
T_Size = 12, n_Transactions = 500000
33% insert 34% contains 33% remove

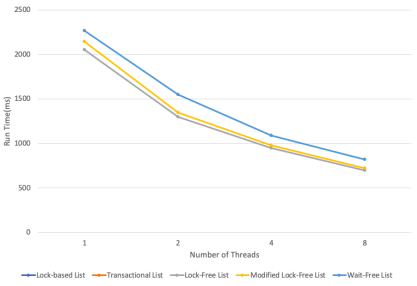


**Figure 12**
T_Size = 100, n_Transactions = 50000
9% insert 90% contains 1% remove

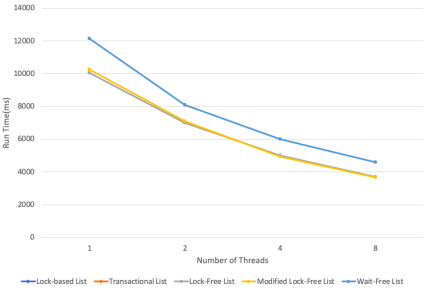


**Figure 13**
T_Size = 80, n_Transactions = 500000
31% insert 37% contains 31% remove

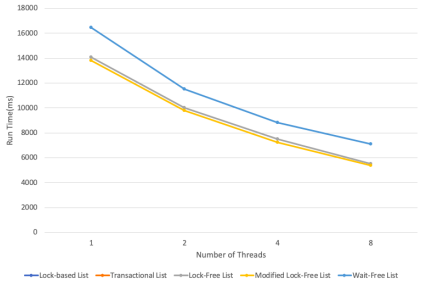


**Figure 14**
T_Size = 100, n_Transactions = 500000
33% insert 34% contains 33% remove

a reference, I applied *TM_CALLABLE* to each of the *insert*, *remove* and *contains* operation to mark them so that they are callable by TM. I also used *TM_BEGIN* and *TM_END* to start and end a transaction when making a call to any of the three operations. I used the "bmharness.cpp" benchmark which directly provides the code for thread management.

## 6 PERFORMANCE EVALUATION

### 6.1 Advantages and Disadvantages of using List

The above list is implemented using a linked list instead of using an array. Linked List is a dynamic data structure, so it can increase or decrease its size at runtime which is not possible with an array in which we need to specify the size in the beginning and in case of the dynamic array also, the resize operation is very costly. Also, an array allocates a block of one contiguous memory, but linked list nodes can allocate its nodes at different locations and only when required. Addition/Removal of nodes in the linked list is efficient as it locates the point of modification and adds the node there. However, in an array, each addition/deletion affects the array elements after the modification point which is a costly operation. Every data structure has its own benefits as well as drawbacks when compared with each other and is suitable for different applications. The disadvantage

of a linked list is that it consumes an extra space storing the next pointer which is not required in an array. Also, searching(contains) operation is time-consuming in a linked list.

Linked List is used in the collision resolution operation of the Hash Tables. It is also used to implement stacks, queues, trees, and graphs. The above described lock-free list is used to implement dynamic size Non-Blocking Hash Tables as it greatly simplifies the task of moving the elements among buckets during the resize operation [10]. The Non-Blocking Hash Tables with Lock-free implementation outperforms the state-of-the-art split ordered list. The Lock-Free List algorithm is extended to remove the head element in the Lock-free pending event set management [7].

## 6.2 Empirical Results

The performance of each of the algorithms is measured on a 4-core single-processor with 4 GB RAM in Ubuntu-Linux (virtual machine). To start the testing, the list is initialized with 128 * n_threads elements where n_threads is the number of threads for that test. Each benchmark run was measured by starting the timer before initializing the linked list and stopping the timer when all threads have finished (join). I conducted various experiments with different transaction sizes(t_size), a number of transactions (n_transactions) and different distribution of operations.

Figures 6, 7 and 8 show the graphs with n_transactions = 10000 with a different distribution of operations and each transaction has a size of one and the insert, contains and remove operation is selected with a probability of 9%, 90%, and 1%. As depicted from Figure 6, the run-time of the lock-free list, modified lock-free list, and wait-free list decreases with the increase in the number of threads. However, the lock-based list which has a single mutex lock on both the insert and remove method calls increases with the increase in n_threads. It is because of the fact that at any given time only a single thread will work because of the coarse-grained locks. Hence, it is basically a sequential linked list and the run-time increases because of the increase in the contention for the mutex lock. Also, the list developed on RSTM performs better than the lock-based list, however, it is not as run-time efficient as lock-free or wait-free list. Another observation from this figure is that if the thread count is 1, then Lock-based List executes in the least amount of time compared to other list implementations because it doesn't use atomic instructions. The wait-free list performs poorly when the thread count is 1 because it uses more *CAS* instruction than the Lock-free list implementation. As we move from figure 6 to 8, the number of lookup operations are reduced, the graph behavior also changes slightly. The run-time of the lock-free list and wait-free list increases from 470 ms to 510ms and 525ms to 580 ms respectively. Also, the gap between the lock-free list and the wait-free list is increased with the increase in the number of threads. It is because of the fact that in a wait-free list, the *insert* operation uses one *FetchAndIncrement* instruction to atomically increment the counter for phase number and four *CompareAndSwap* instruction to enlist the new node at head(3) and to update the state of the new node(1). Also, the *remove* operation uses one *FetchAndIncrement* instruction to atomically increment the counter for phase number and mostly three *CompareAndSwap* instruction to enlist the new node at the head. However, in lock-free list algorithm, the *insert* operation uses only two *CompareAndSwap* instruction - one to enlist the new node at the head and another to update the state of the new node and the *remove* operation also uses only one *CompareAndSwap* instruction to enlist the new node at the head. The *remove* operation also uses one extra *CompareAndSwap* instruction to set the state to *INV* if the found node has state *INS* in both the lists. This result shows that the usage of *CompareAndSwap* instructions is a very costly operation and it should be used only when there is a complete necessity.

In Figures 9, 10 and 11, the number of transactions is fixed, i.e., 500000 but the transaction size is doubled when moving from Figure 9 to 11. As the n_transactions is doubled, the load also increases and the graph is shifted from a scale of 200-900ms to a scale of 800-3000ms after excluding the STMList. The STMList behaves like this because of the increase in the size of a transaction. One of the features of the STM is that it provides compositionality which is not provided by any of the implemented algorithms here. So, if there is something unusual encountered in the STM List, then the whole transaction is repeated again which explains its poor performance. All the other list implementation behaves similarly to the figures above but with the increase in load.

Figure 12, 13 and 14 are shown without the lock-based and STM List to look at the performance of the lock-free and the wait-free list algorithm closely. The results are quite similar and the gap between lock-free and wait-free is less only in the practical scenario of 90% lookup operations. The modified-lock free list performs similarly to the original lock-free list as the contains operation doesn't use any atomic instruction and there is an equal proportion of insert, remove and contains operation. However, we can see that in the 90% look-up operation, the modified-list takes slightly more time than the original list because there are not many nodes with state *INV* (low insertion and deletion) and the extra check of the node *state*.

# 7 CONCLUSION

In this report, a total of five algorithms for the concurrent Unordered list are shown - Lock-based List, Lock-Free List, modified Lock-Free List, Wait-Free List, and STM List. The main algorithms are described in detail with their synchronization techniques to provide concurrency, their progress guarantees, and the correctness conditions. Also, the performance is evaluated on a different distribution of operations, different transaction sizes and a different number of transactions. If only performance is considered, then the Lock-Free List performs the best. However, If composability is required, then the STM algorithm is the only option. As the wait-free list doesn't perform well when the contention is low, an adaptive strategy based on fast-path-slow-path methodology is used by the developers of the above Lock-Free and Wait-Free List to improve the performance of the concurrent list.

# REFERENCES

[1] M. Greenwald. 1999. Non-Blocking Synchronization and System Design. PhD Thesis, Stanford University.
[2] Maurice Herlihy. 1991. Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 13(1). , 124-149 pages.
[3] A. Kogan and E. Petrank. 2011. Wait-free Queues with multiple Enqueuers and Dequeuers. Proceedings of the Sixteenth ACM Symposium on Principles and Practice of Parallel Computing.

[4] Timothy L.Harris. 2001. A practical implementation of Non-Blocking Linked Lists. In Welch, J.L. , 300-314 pages.
[5] M.M. Michael and M.L. Scott. 1996. Simple, Fast, and Practical NonBlocking and Blocking Concurrent Queue Algorithms. In: Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing.
[6] V. Luchangco M. Moir William N. Scherer III S. Heller, M. Herlihy and N. Shavit. 2006. A Lazy Concurrent List-based Set Algorithm. Springer, Heidelberg. , 3-16 pages.
[7] Philip A. Wilsey Sounak Gupta. 2014. Lock-free pending event set management in time warp. *SIGSIM PADS '14 Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (May 2014), 15–26. https://doi.org/may18-21,2014
[8] John D. Valois. 1995. Lock-Free Linked Lists using Compare-and-Swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing.
[9] Christopher Heriot Athul Acharya David Eisenstat William N. Scherer III Michael L. Scott Virendra J. Marathe, Michael F. Spear. 2006. DMCA Lowering the overhead of nonblocking software transactional memory. (2006).
[10] Michael Spear Yujie Liu, Kunlong Zhang. 2014. Dynamic-sized nonblocking hash tables. *Proceedings of the 2014 ACM symposium on Principles of distributed computing* (July 2014), 242–251. https://doi.org/July15-18,2014