GCList: Garbage Collection in Concurrent Sets

Shekhar Bhandakkar, Jonathan Marbaniang and Sathya Peri
Indian Institute of Technology, Hyderabad

{cs14btech11006, cs16mtech11006, sathya_p}@iith.ac.in

June 3, 2018

Abstract

Garbage Collection in concurrent data structures, especially lock-free ones, pose multiple design and conistency challenges. In this instance, we consider the case of concurrent sets. A set is a collection of elements, where the elements are ordered and distinct. These two invariants are always maintained at every point in time.

Sets are usually represented as a linked list of nodes, with each node denoting an element in the Set. Operations on the set include adding elements to the set, removing elements from it and searching for elements in it. Currently, multiple implementations of concurrent sets already exist. LazyList[1], Hand-over-hand List[2] and Harris' List[3] are some of the well-known implementations. However none of these implementations employ, or are concerned with garbage collection of deleted nodes. Instead each implementation ignores deleted nodes or depends on the language's garbage collector to handle them.

Additionally, Garbage collection in concurrent lists, that use optimistic traversals or that are lock-free, is not trivial.

For example, in Lazy List and Harris' List, they allow a thread to traverse a node or a sequence of nodes after these nodes have already been removed from the list, and hence possibly deleted. If deleted nodes are to be reused, this will potentially lead to the ABA problem.[4]

Moreover, some languages like C++ do not have an in-built garbage collector. Some constructs like Shared Pointers[5] provide a limited garbage collection facility, but it degrades performance by a large scale. Integrating Shared Pointers into a concurrent code is also not a trivial task.

In this paper, we propose a new representation of a concurrent set, GCList, which employs in-built garbage collection. We propose a novel garbage collection scheme that implements in-built memory reclamation whereby it reuses deleted nodes from the list. We propose both lock-based and lock-free implementations of GCList. The garbage collection scheme works in parallel with the Set operations.

In our experiments with varying workloads and randomised Set operations, GCList shows comparable performance to LazyList[1] & Harris' List[3] while outperforming Shared Pointers[5], Hazard Pointers[6] and Hand-over-hand List[2]. GCList also consumed 3-4 times less memory as compared to LazyList[1], Hand-over-hand List[2] and Harris' List[3] and is comparable to Shared Pointers[5] and Hazard Pointers[6].

1 Introduction

List-based implementation of concurrent sets are fairly common. LazyList[1], Hand-over-Hand List[2] and Harris's LockFreeList[3] are some common examples. However none of these implementations address the

issue of garbage collection of nodes deleted from the list. Either the algorithm ignores the issue or it relies on the language's garbage collector to handle it for them.

There are several reasons to implement our own memory management scheme. Languages such as C and C++ do not provide garbage collection and often it is more efficient to do our own memory management. C++ has some constructs like shared pointers[5] that offer limited garbage collection facility. Other garbage collection techniques like Stop-the-World are also available. Even though Shared Pointers, Hazard Pointers[6] and these other garbage collection schemes are very generic techniques, since they can be applied to almost all concurrent data structures, they are expensive and cost a lot in terms of performance and the extra data structures required to implement them.

Integrating Shared Pointers, Hazard Pointers and these other garbage collection schemes into a concurrent data structure is also not a trivial task. And more often than not, they are not very optimized for performance. They become even more complicated in case of lock-free data structures employing lock-free methods. Garbage collection, in these cases, is byzantine.[4]

In this paper, we concentrate on the garbage collection scheme for a concurrent set. We introduce a new representation of a concurrent set, **GCList**, with in-built garbage collection. Nodes that are removed from the set are collected in a "Pool" of deleted nodes, to be reused for later add operations. We introduce both lock-based and lock-free versions of GCList. We use the terms node, key and value interchangeably in this paper.

2 System Model & Preliminaries

In this paper, we assume that our system consists of finite set of p processors, accessed by a finite set of n threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking higher-level methods on the shared objects and getting corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail.

Safety: To prove a concurrent data structure to be correct, *linearizability* proposed by Herlihy & Wing [7] is the standard correctness criterion in the concurrent world. They consider a history generated by a data structure which is collection of method invocation and response events. Each invocation of a method call has a subsequent response. A history is linearizabale if it is possible to assign an atomic event as a *linearization point* inside the execution interval of each method such that the result of each of these methods is the same as it would be in a sequential history in which the methods are ordered by their linearization points [7].

Progress: The *progress* properties specifies when a thread invoking methods on shared objects completes in presence of other concurrent threads. Some progress conditions used in this paper are mentioned here which are based on the definitions in Herlihy & Shavit. The progress condition of a method in concurrent object is defined as: (1) Blocking: In this, an unexpected delay by any thread (say, one holding a lock) can prevent other threads from making progress. (2) Deadlock-Free: This is a **blocking** condition which ensures that **some** thread (among other threads in the system) waiting to get a response to a method invocation will eventually receive it. (3) Wait-Free: This is a **non-blocking** condition which ensures that **every** thread trying to get a response to a method, eventually receives it[8].

3 Related Work

We discuss some of the list-based set algorithms in this section and some existing garbage collection techniques that can be used in concurrent sets.

3.1 Hand-Over-Hand List

In this list-based representation of a set, also called **lock-coupling**[2], each thread traverses the list from the head of the list, while acquiring fine-grained locks in a hand-over-hand manner. Each thread acquires the lock for the next node and then releases the lock for the current node.
All operations require the usage of locks which may affect the overall performance of the list, even though garbage collection in this list is a fairly trivial task.

3.2 LazyList

An improvement over the Hand-over-Hand list is the LazyList[1]. Threads traverse the list **optimistically**, without using any locks. Nodes are locked only when the required pair are found. An additional boolean field called "marked" field is associated with every node. The "marked" field is used to identify nodes that have been deleted but are still reachable from the head of the list.

In LazyList, nodes are deleted in two steps:
- **Logical deletion:** The marked field is set to true.
- **Physical deletion:** The node's predecessor's next reference is swung to the node's successor.

The contains method is completely wait-free. It traverses the list without using any locks. It's easy to see that garbage collection, in this case, is not so trivial. It may lead to an issue known as the **"ABA Problem"**[4].
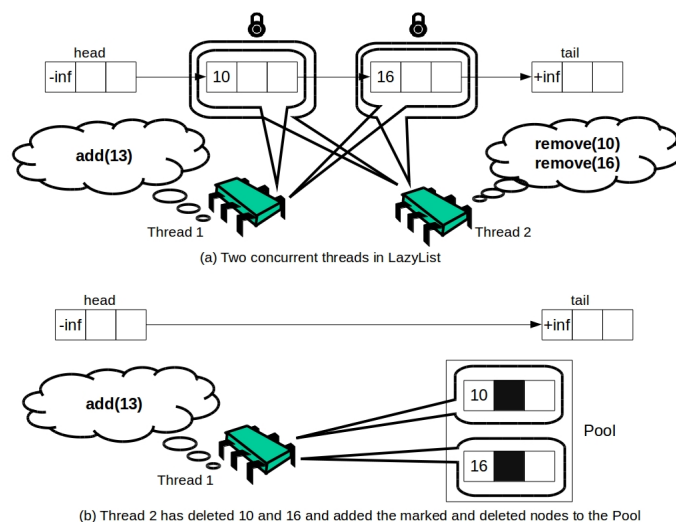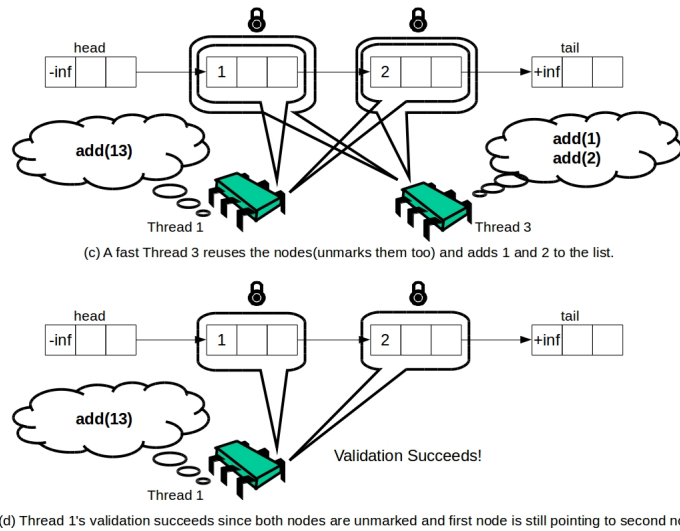

(a) Two concurrent threads in LazyList

(b) Thread 2 has deleted 10 and 16 and added the marked and deleted nodes to the Pool

Figure 1: The ABA Problem in LazyList (Part 1)

(c) A fast Thread 3 reuses the nodes(unmarks them too) and adds 1 and 2 to the list.

(d) Thread 1's validation succeeds since both nodes are unmarked and first node is still pointing to second node

Figure 2: The ABA Problem in LazyList (Part 2)

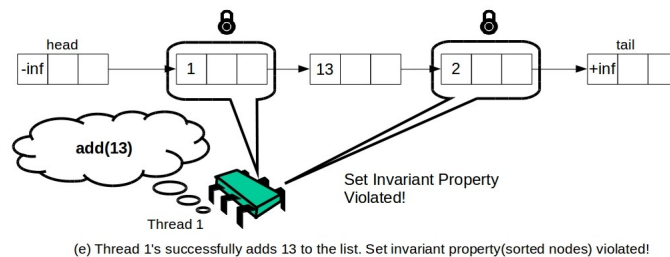(e) Thread 1's successfully adds 13 to the list. Set invariant property(sorted nodes) violated!

Figure 3: The ABA Problem in LazyList (Part 3)

3.3   LockFreeList

The LockFreeList[9] is an extension of the LazyList[1], where locks are eliminated altogether from the list operations and all the methods are non-blocking[8].

The list uses an AtomicMarkableReference[10] object as a part of it's structure, which allows a thread to atomically read and update both the boolean mark and the next reference of a node. The list also uses compareAndSet or CAS calls for its operations.

The remove method is similar to LazyList[1], in that deletion is done in two steps.
- A CAS call is used to set the marked field of a node.
- Another CAS call is used on the node's predecessor to physically delete the node from the list.

An important difference between LockFreeList[9] and LazyList[1] is that LockFreeList never traverses logically marked nodes. Instead the encountered marked nodes are physically deleted from the list. Essentially, threads "help" out other slower threads that have completed the first CAS call but not the second.

It can also be seen that similar to LazyList[1], LockFreeList[9] is also vulnerable to the ABA problem[4].

3.4 Reference Counting

In a dynamic and concurrent data structure, arbitrary objects can continuously and concurrently be added or removed from the data structure. And multiple owners may have a reference to the shared objects. Unsafe freeing of a node may lead to safety issues and possible crashes[11].
So, before freeing a shared object, it should be checked that there are no remaining references to it. This should also include possible local references to the shared object that any thread might have, as a read or write access to the memory of a reclaimed object might be fatal to the correctness of the data structure andor to the whole system[11].

In the "Reference-Counting" category of garbage collection techniques, shared counters are assigned to objects and they are used to count the number of references to any object at any given time[11]. In other words, a group of owners share the ownership for an object.This group is responsible for deleting that object when the last one among them releases that ownership. The shared object can be freed if and only if the counter becomes zero[12].

This method, however is expensive. A shared atomic counter has to be associated with every object[11][12]. Getting a reference to an object and incrementing the shared counter has to be an atomic operation. Same thing applies when losing the reference to the object and decrementing the shared counter. Even a simple read operation from the shared object has to increment the shared counter. Essentially, the memory read becomes a read-modify-write operation[13].

3.5 Pointer-based techniques

Pointer-based techniques such as Hazard Pointers[6] explicitly mark live objects (objects that threads can access) which are not de-allocated. Pointer-based schemes suffer from two limitations: they must be customized to the data structure at hand, which makes them difficult to deploy; they publish each pointer that is used in a shared memory location, which is expensive in terms of synchronization.

Hazard Pointers (HP) and other pointer-based techniques will typically publish the pointer to each object they use, and then check that the pointer has not changed in the meantime. Such approach guarantees that an object which has been deleted will not be later de-referenced, at the cost of each reader doing synchronization on a per-object basis.

Because it requires validation of the pointer that will be accessed next, Hazard Pointers are lock-free for readers, although in some situations they can be made wait-free for readers. HP is wait-free bounded for reclamation, with the bound being proportional to the number of threads times the number of hazard pointers, because each reclaimer has to scan all the hazard pointers of all the other threads before deleting a node. In HP the retired nodes are placed in a retired list which is scanned once its size reaches an R threshold. In terms of memory usage, when the R factor is set to the lowest setting of 1, each reclaimer can have at most a list of retired nodes with a size equal to the number of threads minus 1, times the number of hazard pointers. If each thread has one such list of nodes pending to be deleted, at any given point in time there are at most $O(N_{threads}^2)$ nodes to be deleted.

4 Our Algorithm: GCList

This paper introduces the **GCList**, a list-based set algorithm with an in-built garbage collection scheme. The set is represented as a linked-list of nodes, supporting the following operations:
- **add(key)**, adds key to the set, and returns true if and only if key was not already present in the set.
- **remove(key)**, removes key from the set, and returns true if and only if key was present in the set.
- **contains(key)**, searches for key in the set, and returns true if and only if key is present in the set.
We introduce two versions of GCList, a blocking version or **GCLBList** and a non-blocking version or **GCLFList**.

The actual code for both the versions has been kept in the appendix. The code is in C++.

4.1 GCLBList

Each node in the list consists of three fields: the key field, an **AtomicStampedReference**[14] object called as infoNext and a lock associated with the node. We have implemented our own AtomicStampedReference[14] in C++. The list is ordered according to the keys of each node. infoNext contains a reference to the next node in the list and an integer stamp associated with the node. Both the stamp and the reference can be read and updated atomically[14]. The lock field is a lock used for synchronization.

```cpp
class Node
{
  int key;
  AtomicStampedReference<Node> infoNext;
  mutex nodeLock;
};
```

GCList Node

(a) Shaded area represents the key value

GCList Node

(b) Shaded area represents the **AtomicStampedReference** Object, called infoNext, consisting of an integer stamp and a next reference. The stamp and the reference can be read and updated atomically.
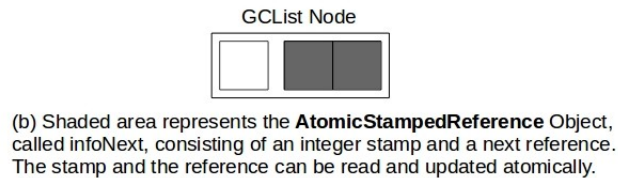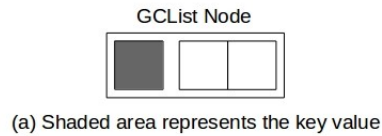
Figure 4: GCList Node and it's components

As mentioned earlier, we consider three operations on the list i.e. add, remove and contains. However we factor out functionality common to the add and remove methods by creating an inner Window class to help navigation. The common functionality is used to optimistically traverse the list and "find" the required pair of nodes required for each operation. The find method then returns the references to the nodes and their respective stamps in a Window object to the calling method.

```cpp
class Window
{
  public:
    Node *pred, *curr;
    int predSt, currSt;
    Window(Node *myPred, Node *myCurr, int myPredSt, int myCurrSt)
    {
      pred = myPred;
      curr = myCurr;
      predSt = myPredSt;
      currSt = myCurrSt;
    }
};
```

4.1.1 The find method

The find method is used by the add and remove methods to optimistically traverse the list. The thread gets a reference to the "head" node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the infoNext's reference and stamp fields of a node are read atomically[14]. The thread keeps traversing the list until it finds the relevant pair of nodes, pred and curr. curr holds a reference to the first node with a key greater than or equal to the key that is being searched, in the list, with pred being curr's predecessor. The find method returns a window object, containing references to pred and curr along with their respective stamps, to the calling method.

An important observation to be made here is the use of stamps during traversal. Stamps are used to detect synchronization conflicts by a traversing thread. This can be inferred from the working of the remove method later. If at any time during a thread's traversal, the stamp of the pred node changes, a synchronization conflict with another "removing" thread is detected. The current thread "retries" it's traversal from the head node.

4.1.2 The validate method

The validate method is used to ensure that the calling method has locked the correct pair of nodes. It uses the stamps and references returned by the find method to ensure that both pred and curr are still present in the list and pred is still pointing to curr. If the stamps of either node has changed or pred is no longer pointing to curr, then it signifies a synchronization conflict with another thread. The current thread then restarts it's execution.

4.1.3 The remove method

The remove method is used to remove key from the set, returning true if and only if key was in the set. It calls the "find" method to determine the correct pair of nodes for the remove operation. The nodes are locked and then validation is performed using the "validate" method. If validation fails, the nodes are unlocked and the thread retries, otherwise it continues it's operation.

Deletion is performed in two steps:

- **Step 1:** pred's infoNext's reference is swung to curr's infoNext's reference and pred's infoNext's stamp is incremented by one. This operation to update pred's infoNext's reference and stamp fields is atomic.

- **Step 2:** curr's infoNext's stamp is incremented by 1. This marks the successful deletion of curr from the list.

Step 2 is the linearization point for the remove method.

After curr has been successfully deleted, it is added to the "Pool". A **Pool** is a concurrent data structure which is used to hold the deleted nodes. These deleted nodes can now be reused for later add operations.

Now, an important thing to discuss in this section is why does a thread traversing the list, in the find or contains method, has to retry if the pred's stamp changes. Based on the working of the "find" method, we can see that if any thread has a reference to curr, it should also have read pred's old stamp. This is because reads from an AtomicStampedReference[14] object is atomic. At this point, if curr were to be deleted from the list, pred's stamp would have been incremented, in Step 1. Again, this updation of pred's infoNext fields is atomic.

If the current thread were to continue it's traversal, it may instead traverse the Pool or some other part of the list, since we have no guarantees about curr's position after it's deletion. Instead, before advancing pred and curr in the list, we check pred's stamp again. If it has changed, it implies that curr may have been deleted and the current thread is in a synchronization conflict with a removing thread. The current thread then restarts it's traversal from the list's head again. If pred's stamp is unchanged though, it implies that curr is still a part of the list and the thread can advance pred and curr.

Conversely, we can say that if a thread has read pred's updated stamp at the first read, then it cannot have a reference to curr. Again, this is because the updation of pred's infoNext fields is atomic[14].
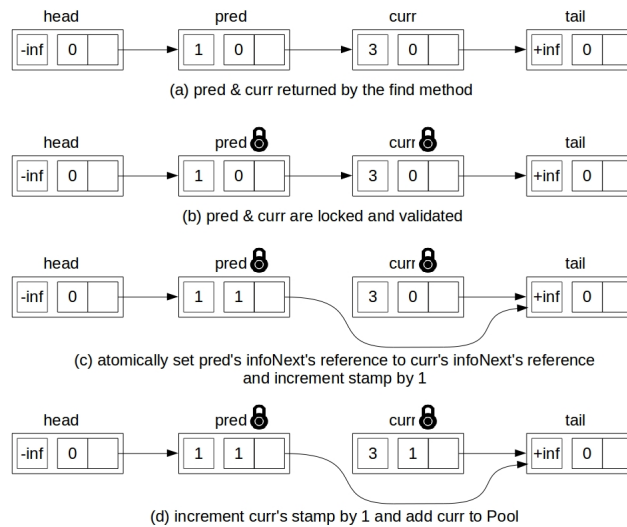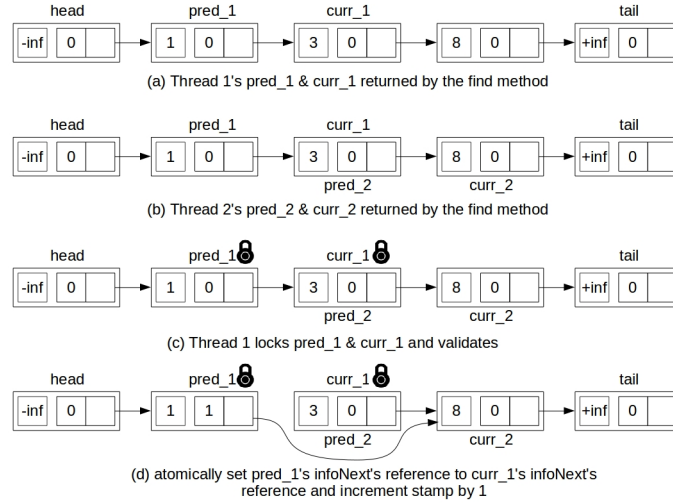

Figure 5: GCLBList: Remove Steps

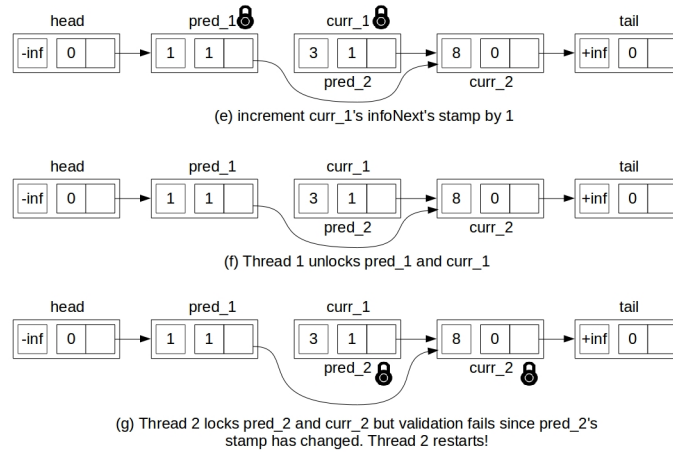Figure 6: GCLBList: Two concurrent removing Threads(Part 1)

Figure 7: GCLBList: Two concurrent removing Threads(Part 2)

4.1.4 The add method

The add method is used to add a key to the list if and only if the key is not already present in the list. It calls the "find" method to determine the correct pair of nodes for the add operation. The nodes are locked and then validation is performed using the "validate" method. If validation fails, the nodes are unlocked and the thread retries, otherwise it continues it's operation. The thread then queries the Pool(a data structure containing deleted nodes) for a node. If the Pool is not empty, a node is returned to be reused. Else, the thread creates a new node. It then inserts the new node, unlocks pred and curr and returns true.

The step in which pred's infoNext's reference is set to the new node is the linearization point for the add method.
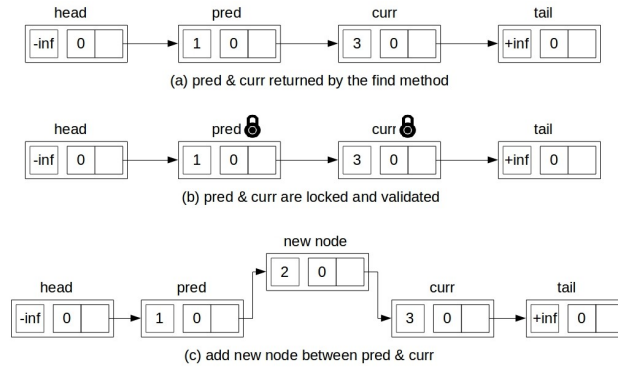
Figure 8: GCLBList: Add Steps

4.1.5 The contains method

The contains method is similar to the find method. It starts from the "head" node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the infoNext's reference and stamp fields of a node are read atomically[14]. The thread keeps traversing the list until it finds the first node with a key greater than or equal to the key that is being searched.

Similar to the find method, stamps are used to detect synchronization conflicts during traversal. If at any time during a thread's traversal, the stamp of the pred node changes, a synchronization conflict with another "removing" thread is detected. The current thread "retries" it's traversal from the head node.

The method returns true if and only if the key is present in the list.

4.2 GCLFList

GCLFList is the non-blocking version of our list-based set algorithm.

Each node in the list now consists of two fields, the key field and an AtomicStampedReference[14] object called as infoNext. The list is ordered according to the keys of each node. infoNext contains a reference to the next node in the list and an integer stamp associated with the node. Both the stamp and the reference can be read and updated atomically[14]. There is no lock field associated with the node anymore.

```
class Node
{
public:
  int key;
  AtomicStampedReference<Node> infoNext;
}
```

We instead use atomic functions like compareAndSet[14] or CAS to perform our operations on the list. Atomic operations[14] are used to atomically read and update the AtomicStampedReference object associated with each node. However, this also leads to complications. For example, if we follow the deletion steps

of GCLBList, what happens in the case of two adjacent concurrent remove operations, using CAS? We can see that one of the nodes won't be removed from the list.

```
booean CAS(expected reference, new reference, expected stamp, new stamp
    )
{
  if both reference & stamp are as expected, then:
    update reference & stamp to new
    values;
    return true;
  else:
    return false;
    end if
}
```
<div align="center">Listing 1: CAS pseudo-code</div>

To solve this problem, we need a way to identify a marked node in the list, even though it may still be present in the list i.e. a logically deleted node. We differentiate between a logically deleted node and a node that is a part of the list by using parity of stamp.
- A node with an **even stamp** is a part of the list.
- A node with an **odd stamp** denotes a node that has been deleted from the list.

The deletion operation is also divided into two steps
- **Logical Deletion:** Increment curr's stamp by 1 using CAS[14] i.e marking curr.
- **Physical Deletion:** Swing pred's infoNext's reference to curr's infoNext's reference and increment pred's infoNext's stamp by 2, atomically using CAS[14].

We also adopt the concept of **Helping** i.e. if a traversing thread encounters a logically deleted or marked node, it attempts to first remove the node from the list, before advancing forward.
The methods are similar to GCLBList with a few modifications to incorporate the above changes.

4.2.1 The find method

The find method is used by the add and remove methods to optimistically traverse the list. The thread gets a reference to the "head" node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the next reference and stamp of a node is read atomically[14]. The thread keeps traversing the list until it finds the relevant pair of nodes, pred and curr. It returns a window object, containing references to pred and curr along with their respective stamps, to the calling method.

As mentioned above, each time the thread encounters a marked node i.e. a node with an odd stamp, it attempts to physically delete the node first before advancing. If the CAS operation for the physical deletion succeeds, the node advances forward. Else it retries. Threads never traverse marked nodes because they lead to consistency issues.

For example, find may return a marked pred and an unmarked curr to the remove method trying to add a new node between pred and curr. If pred is physically removed by another thread before the new node could

be added, the new node would end up being not added to the list. This difficulty arises because the current thread is not holding locks on pred and curr.

Similar to the previous find method, stamps are also used to detect synchronization conflicts by a traversing thread. If at any time during a thread's traversal, the stamp of the pred node changes, a synchronization conflict with another "removing" thread is detected. The current thread "retries" it's traversal from the head node.

4.2.2 The remove method

The remove method is used to remove key from the set, returning true if and only if key was in the set. It calls the "find" method to determine the correct pair of nodes for the remove operation.
Deletion of "curr" is performed in two steps as mentioned earlier. The step for logical deletion of "curr" is the linearization point for the remove method.
After curr has been successfully deleted, it is added to the "Pool". These deleted nodes can now be reused for later add operations.

Now, what happens if any of the two CAS operations fail.
Case 1: CAS for logical deletion of curr fails. It implies that some other thread has performed a concurrent operation on curr and a synchronization conflict is detected. The current thread has to restart it's operation.
Case 2: CAS for physical deletion fails. It implies that some other thread has performed a concurrent operation on pred. The current thread has two choices: it can depend on other traversing threads to "help" physically delete curr or it can traverse the list once more time to ensure curr's deletion.
An important note is that incrementing the pred's stamp by 2 during physical deletion prevents the ABA problem.
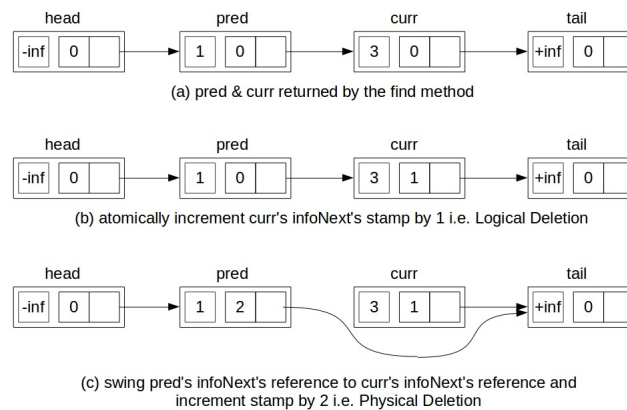

(a) pred & curr returned by the find method

(b) atomically increment curr's infoNext's stamp by 1 i.e. Logical Deletion

(c) swing pred's infoNext's reference to curr's infoNext's reference and increment stamp by 2 i.e. Physical Deletion

Figure 9: GCLFList: Remove Steps

4.2.3 The add method

The add method is used to add a key to the list if and only if the key is not already present in the list. It calls the "find" method to determine the correct pair of nodes for the add operation. The thread then queries the Pool for a node. If the Pool is not empty, a node is returned to be reused. Else, the thread creates a new
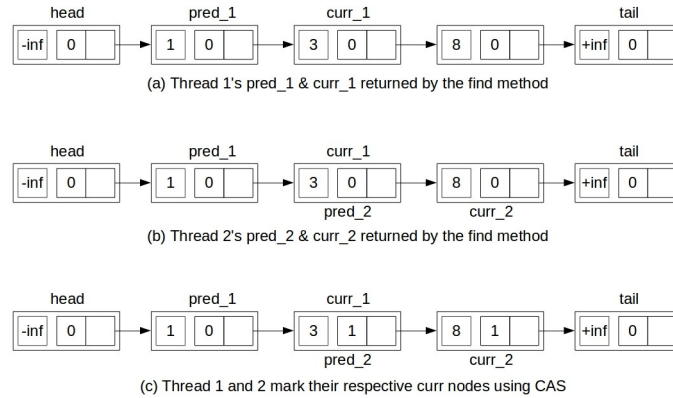
(a) Thread 1's pred_1 & curr_1 returned by the find method

(b) Thread 2's pred_2 & curr_2 returned by the find method

(c) Thread 1 and 2 mark their respective curr nodes using CAS

Figure 10: GCLFList: Two concurrent removing Threads(Part 1)

(d) Thread 1 physically deletes curr_1 by executing CAS on pred_1.
Thread 2's CAS for physical deletion of curr_2 fails!
curr_1 is added to the "Pool".

(e) curr_2 is still a part of the list but it has been marked.

(f) curr_2 physically deleted by a "helping" thread or by
Thread 2 retraversing the list.
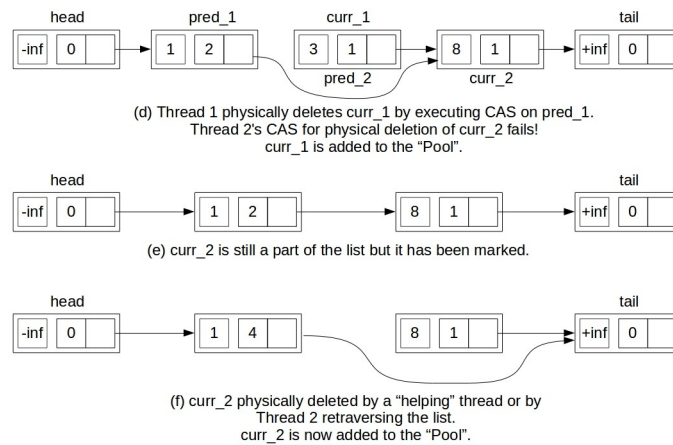curr_2 is now added to the "Pool".

Figure 11: GCLFList: Two concurrent removing Threads(Part 2)

node. It then inserts the new node, unlocks pred and curr and returns true. If the node is obtained from the Pool, it's stamp is incremented by 1 before inserting it into the list.

An important observation to be made her is if the adding thread's CAS call on pred to insert the new node to the list fails, it calls the find method again, resulting in a new pair of pred and curr. However, another concurrent adding thread may have meanwhile added the same key to the list. The current thread now cannot add the same key anymore and has to return false. Before doing that, if the node was retrieved from the Pool, it is added back again to it. Else, if it was a newly created node, we can delete it since we have a guarantee that no other thread has a reference to it.

This scenario never occured in GCLBList since once pred and curr were locked and validated and the key was previously absent from the list, there is a guarantee that the current thread would be able to add the key to the list successfully. Provided it doesn't crash midway before that.
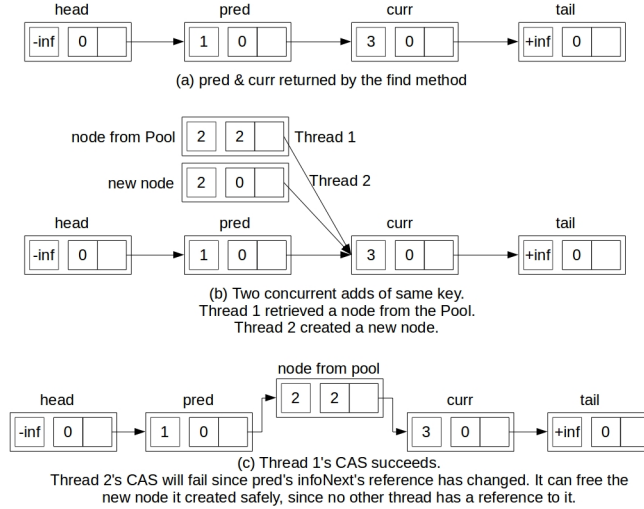The CAS call to set pred's infoNext's reference to the new node is the linearization point for this method.

Figure 12: GCLFList: Add Steps

4.2.4 The contains method

The contains method starts from the "head" node and keeps traversing the list in an optimistic hand-over-hand fashion. At every step of the traversal, the next reference and stamp of a node is read atomically[14]. The thread keeps traversing the list until it finds the first node with a key greater than or equal to the key that is being searched.

Again, stamps are used to detect synchronization conflicts during traversal. If at any time during a thread's traversal, the stamp of the pred node changes, a synchronization conflict with another "removing" thread is detected. The current thread "retries" it's traversal from the head node.
The method returns true if and only if the key is present in the list and it's infoNext's stamp is even.

5 The Pool

The pool is a concurrent data structure that is used to hold the deleted nodes that have been reclaimed from the list. In our experiments we used two different queue implementations to act as the pool. The code for both the queues has been kept in the appendix.

5.1 The blocking unbounded total queue

This lock-based concurrent queue[9] uses two separate locks for each queue operation i.e. an enqLock to enqueue a deleted node to the queue and a deqLock to dequeue a node from the queue, respectively.

Before a thread performs an enqueue or a dequeue operation, it acquires the corresponding lock on the queue. After acquiring the lock, the thread performs it's operation and releases the lock upon completion. The lock ensures that, at a particular time, only one thread is able to perform an enqueue or a dequeue operation on the queue.

5.2 The unbounded Lock-free queue

This lock-free concurrent queue[9] uses atomic compareAndSet or CAS calls instead of locks for the queue operations. The CAS calls are used to enqueue a node into the queue and also to dequeue a node from the queue.

This lock-free implementation helps to prevent faster threads from starving, with the removal of coarse-grained locks. This queue implementation also uses the concept of helping, where faster thread help the slower threads to finish their queue operations.

The enqueue operation is done in two steps:
- The thread locates the last node in the queue and uses a CAS call to append the new node to the queue.
- It then uses another CAS call to change the queue's tail from the previous last node to the current last node.

Since the above two CAS calls are not a single atomic operation, threads help each other to complete the second CAS, if a half finished enqueue operation is encountered.

An important attribute to be noted about the queue is that it also uses the AtomicStampedReference[14] object, in its' head and tail, to prevent the ABA problem[4] problem from occurring in the queue.

6 Evaluation

We tested both versions of GCList against existing implementations of a concurrent set namely: LazyList[1], Hand-over-Hand List[2], Harris's LockFreeList[3] and a Shared Pointer[5] version of LazyList.

We used both the lock-based queue and the lock-free queue, as a Pool, in combination with the two versions of GCList. The resultant set representation is named by using the list's name as prefix and pool's name as suffix. For example, the GCLBList using the lock-based queue would be named GCLBListLBQueue and the GCLFList using the lock-free queue would be named GCLBListLFQueue.
The LazyList based on Shared Pointers has been named LazyList_SP. The LockFreeList using Hazard Pointers for Memory Reclamation has been named LockFreeList_HP.

We tested the above mentioned algorithms versus our algorithms for both performance and memory consumption, with varying workloads and randomized Set operations.

For performance, we fix the total number of operations that each thread can perform, divided in varying ratios between adds, removes and contains. We allowed each algorithm to run for 10 seconds and measured the number of operations completed during said time period.
For memory consumption, we fix the total number of operations that each thread can perform, divided in varying ratios between adds, removes and contains. We keep track of the number of times each thread allocates and deallocates memory. Whenever the thread allocates new memory, a thread-local variable is incremented and whenever the memory is released, the variable is decremented. At the end of all thread operations, the main thread consolidates the sum of all the thread-local variable. We consider this node count as the memory consumed by the algorithm during it's entire execution.

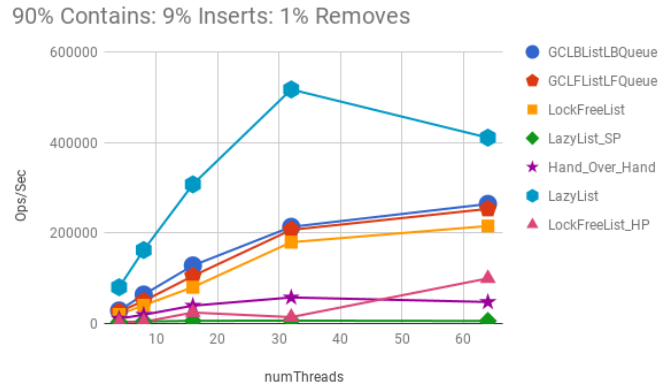Based on the above criteria, we obtained the following graphs.

Figure 13: Performance Evaluation: 10% writes
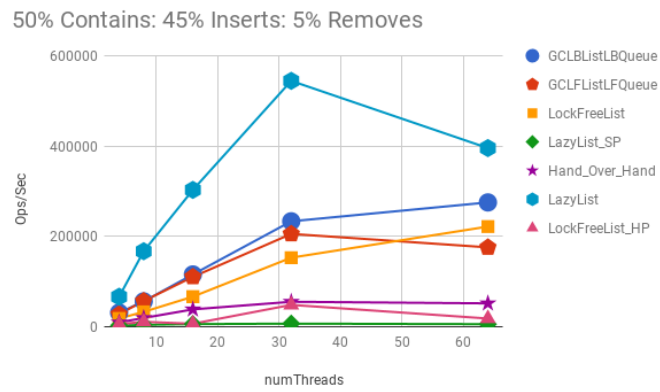

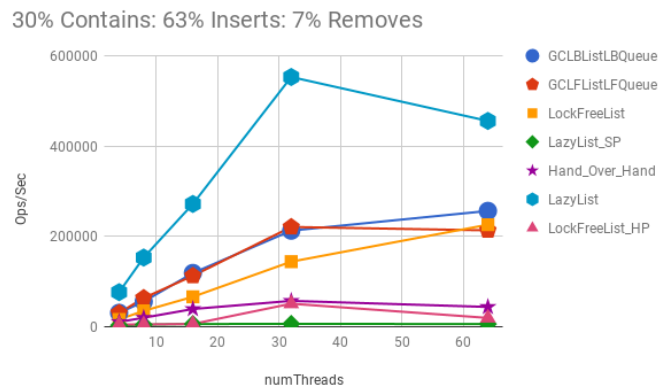Figure 14: Performance Evaluation: 50% writes

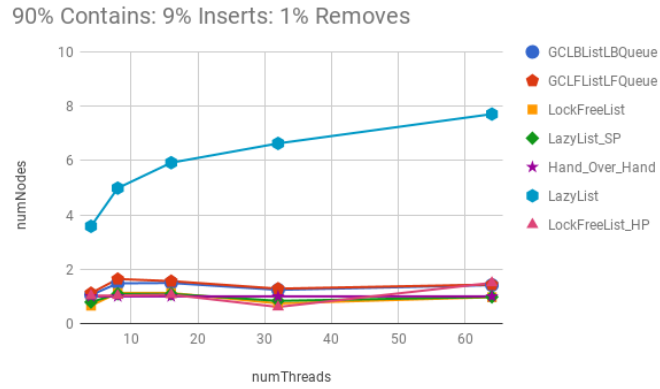Figure 15: Performance Evaluation: 70% writes
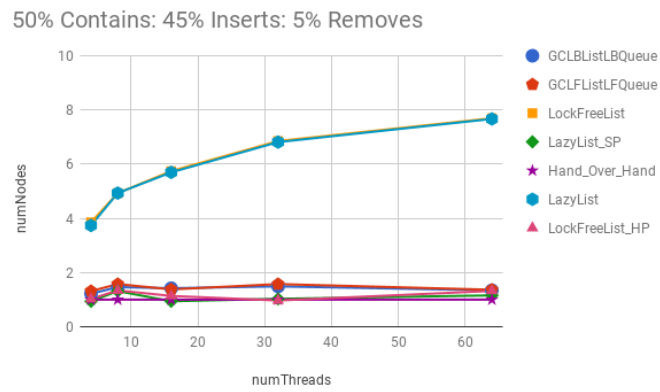
Figure 16: Memory Consumption Evaluation: 10% writes

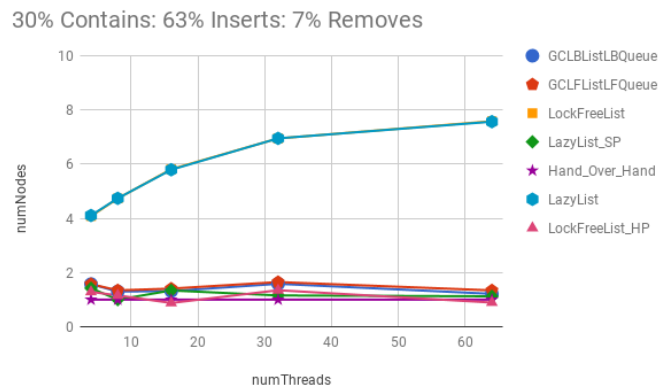Figure 17: Memory Consumption Evaluation: 50% writes

Figure 18: Memory Consumption Evaluation: 70% writes

6.1 Analysis of Results

6.1.1 Performance Analysis

From the above graphs, we can see that the performance of both versions of GCList i.e GCLBList and GCLFList, is at par or even better than Harris's LockFreeList. Both outperform the Hand-over-Hand List, the LazyList based on Shared Pointers and the LockFreeList using Hazard Pointers for memory reclamation by multiple folds. The GCList versions are only outperformed by the original LazyList.

6.1.2 Memory Consumption Analysis

However, in terms of Memory consumption, both versions of GCList consume a lot less memory than the original LazyList. It also needs less memory than Harris's LockFreeList and the Hand-over-Hand List. In comparison with generic techniques like Shared Pointers and Hazard Pointers, memory consumption of GCList is still comparable to both.

7   Conclusion and Future Work

In this paper, we have presented **GCList**, a linked-list representation of a concurrent set, with in-built garbage collection. Both the lock-based and lock-free versions of **GCList** are introduced.

Our results show that **GCList** matches or outperforms most of the existing representations of a concurrent set, while consuming a lot lesser memory than the high-performing algorithms like LazyList. Memory consumption was at par with a generic garbage collection facility like Shared Pointers, while outperforming them many folds.

In future work, we plan to investigate whether we can extend it to other data structures similar to a concurrent list or using it as a part of it's structure e.g. SkipList, Hash Tables etc.

References

[1] Steve Heller et al. "A Lazy Concurrent List-Based Set Algorithm". In: *OPODIS'05 Proceedings of the 9th international conference on Principles of Distributed Systems* (December 12-14,2005), pp. 3–16. DOI: 10.1007/11795490_3.

[2] R. Bayer and M. Schkolnick. "Concurrency of operations on B-trees". In: *Acta Informatica* 9 (1977), pp. 1–21. DOI: https://doi.org/10.1007/BF00263762.

[3] Harris T.L. " A Pragmatic Implementation of Non-blocking Linked-lists". In: *Proceedings of the Symposium on Distributed Computing (DISC)*. Berlin, Heidelberg, 2001.

[4] M. M. Michael and M. L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (1996), pp. 267–275. DOI: 10.1145/248052.248106.

[5] *Shared Pointers*. 1.http://www.cplusplus.com/reference/memory/shared_ptr/.

[6] M. M. Michael. "Hazard pointers: Safe memory reclamation for lock-free objects." In: *Parallel and Distributed Systems, IEEE Transactions on 15* (2004), 491???504.

[7] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. ISSN: 0164-0925.

[8] Maurice Herlihy. "Wait-free Synchronization". In: *ACM Trans. Program. Lang. Syst.* 13.1 (Jan. 1991), pp. 124–149. ISSN: 0164-0925. DOI: `10.1145/114005.102808`. URL: `http://doi.acm.org/10.1145/114005.102808`.

[9] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375, 9780123977953.

[10] *AtomicMarkableReference Java SE.* `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/atomic/AtomicMarkableReference.html`.

[11] Anders Gidenstam et al. "Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 20.8 (Aug. 2009), pp. 1173–1187.

[12] David L. Detlefs et al. "Lock-free reference counting". In: *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC).* New York, 2001, pp. 190–199.

[13] Dan Alistarh et al. "ThreadScan: Automatic and Scalable Memory Reclamation". In: (June 2015).

[14] *AtomicStampedReference Java SE.* `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/atomic/AtomicStampedReference.html`.

A   Appendices

A.1   GCLBList

A.1.1   The find method

C++ Code.

```
/*
 * find method traverses the list from the head and
 * searches the list for the first node with key whose
 * value is greater than or equal to the required key "k".
 * returns a window object of the said node, node's predecessor
 * along with their stamps.
 */
Window* find(Node *head, int k)
{
  Node *pred, *curr, *succ;
  int predSt, currSt, nPredSt;

  bool breakTest = false;

  if((head->infoNext).getReference() == tail) {
```

```
        return new Window(head, tail, (head->infoNext).getStamp(),(tail->
            infoNext).getStamp());
    }

retry:
    while (true) {
        pred = head;
        /*
         * Start from the head.
         */

        curr = (pred->infoNext).get(&predSt);
        /*
         * Read node's infoNext's reference & stamp atomically.
         */

        while (true) {
            breakTest = (k <= curr->key);
            /*
             * Break when key greater than or equal to required key is found.
             */

            succ=(curr->infoNext).get(&currSt);
            /*
             * atomically read curr's infoNext's reference(successor) and
         stamp.
             * Successor may be null if curr has been deleted.
             */

            nPredSt = (pred->infoNext).getStamp();
            /*
             * Read pred's stamp again before advancing forward.
             * This is the safety check to ensure we are traversing the list
             * correctly, in increasing order of keys.
             */

            if (predSt != nPredSt)
                goto retry;
            /*
             * If pred's new stamp is different from the one read previously,
             * a synchronization conflict is detected.
             * curr may have been deleted by another thread from the list.
             * The thread restarts it's traversal to ensure correctness.
             * If pred's stamp is still the same, then everything is fine.
             */
```

```
    if (breakTest)
      break;
    /*
     * If pred's stamp has not changed, everything is fine.
     * Check if required pair of nodes has been found.
     * If yes, break. Else, continue.
     */

    pred = curr;
    curr = succ;
    predSt = currSt;
    /*
     * Keep advancing pred and curr in the list.
     * Keep track of new pred's old stamp to be used later,
     * to detect synchronization conflicts.
     */
  }
  return new Window(pred, curr, predSt, currSt);
  /*
   * return pred and curr, along with their stamps.
   * encapsulated in a window object.
   */
  }
}
```

A.1.2 The validate method

C++ Code.

```
/*
 * validate is used by the calling method to ensure
 * the consistency of the locked nodes.
 * Takes the node's references and old stamps as arguments.
 * And checks whether any other thread has performed any
 * conflicting operations on them.
 */
bool validate(Node *pred, int predSt, Node *curr, int currSt)
{
  Node *nCurr;
  int nPredSt, nCurrSt;

  nCurr = (pred->infoNext).get(&nPredSt);
  /*
   * read pred's infoNext's reference and stamp atomically.
```

```
  */

  nCurrSt = (curr->infoNext).getStamp();
  /*
  * read curr's infoNext's stamp atomically.
  */

  return(predSt == nPredSt && currSt == nCurrSt && curr == nCurr);
  /*
  * checks if pred is still pointing to curr.
  * and if any of their stamps have changed from their old values.
  * if yes, a conflict is detected.
  * Returns true or false to calling method.
  */
}
```

A.1.3 The remove method

C++ Code.

```
/*
* remove a node with key value "k" from the list.
* uses the find method to get the node with key value greater than or
  equal to k,
* and it's predecessor.
* uses the validate method to ensure consistency of both nodes.
* removes the node, if present, from the list in two steps.
* unlocks the nodes and returns true only if key was present to be
  removed.
* else returns false.
*/
bool remove(int k)
{
  while (true) {
    Window *window=find(head,k);
    /*
    * call find method to traverse list optimistically.
    * gets pred and curr, along with their stamps, in a window object
    */

    Node *pred = window->pred, *curr = window->curr;
    int predSt = window->predSt, currSt = window->currSt;
    /*
    * retrieve pred and curr, and their stamps, from the window object.
    */
```

```
 pred->lock();
 if (!(curr->try_lock())) {
   pred->unlock();
   continue;
 }
 /*
 * lock both the nodes.
 * tryLock() is to prevent deadlocks, since there is no guarantee,
 * that keys are being locked in increasing order.
 */

 if (validate(pred, predSt, curr, currSt)) {
 /*
 * use validate to ensure the consistency of pred and curr.
 */
   if (curr->key != k) {
     curr->unlock();
     pred->unlock();
     return false;
     /*
     * if key is not present, unlock both nodes.
     * and return false.
     */
   } else {
     Node *temp;
     int stamp;

     stamp = (pred->infoNext).getStamp();
     temp = (curr->infoNext).getReference();
     (pred->infoNext).set(temp,++stamp);
     /*
     * Deletion Step 1:
     * atomically swing pred's infoNext's reference to curr's
infoNext's reference
     * and increment pred's infoNext's stamp by 1.
     */

     temp = (curr->infoNext).get(&stamp);
     (curr->infoNext).set(temp,++stamp);
     /*
     * Deletion Step 2:
     * atomically increment curr's infoNext's stamp by 1.
     */
```

```
        Pool->set(curr);
        /*
         * add deleted node "curr" to the Pool.
         * curr can be reused for later add operations.
         */

        curr->unlock();
        pred->unlock();
        return true;
        /*
         * unlock pred and curr.
         * return true.
         */
      }
    }
    curr->unlock();
    pred->unlock();
  }
}
```

A.1.4   The add method

C++ Code.

```
/*
 * add a node with key value "k" to the list.
 * uses the find method to get the node with key value greater than or
    equal to k,
 * and it's predecessor.
 * uses the validate method to ensure consistency of both nodes.
 * adds the node, if not already present, to the list in two steps.
 * unlocks the nodes and returns true only if key was successfully added
    .
 * else returns false.
 */
bool add(int k)
{
  while (true) {
    Window *window = find(head, k);
    /*
     * call find method to traverse list optimistically.
     * gets pred and curr, along with their stamps, in a window object
     */

    Node *pred = window->pred, *curr = window->curr;
```

```cpp
int predSt = window->predSt, currSt = window->currSt;
/*
 * retrieve pred and curr, and their stamps, from the window object.
 */

pred->lock();
if (!(curr->try_lock())) {
  pred->unlock();
  continue;
}
/*
 * lock both the nodes.
 * tryLock() is to prevent deadlocks, since there is no guarantee,
 * that keys are being locked in increasing order.
 */

if (validate(pred, predSt, curr, currSt)) {
/*
 * use validate to ensure the consistency of pred and curr.
 */
  if (curr->key == k) {
    curr->unlock();
    pred->unlock();
    return false;
    /*
     * if key is already present, unlock both nodes.
     * and return false.
     */
  } else {
    Node *node = Pool->get();
    /*
     * query the Pool for a node.
     */

    if (node != nullptr) {
      node->key = k;
      /*
       * node has been retrieved from pool.
       * reuse for new add operation.
       */
    } else {
      node = new Node(k);
      /*
       * pool is empty.
       * create new node.
```

```
      */
    }


    int stamp = (node->infoNext).getStamp();
    (node->infoNext).set(curr,stamp);
    /*
     * set new node's reference to curr.
     * no need to change new node's stamp.
     */


    stamp = (pred->infoNext).getStamp();
    (pred->infoNext).set(node,stamp);
    /*
     * atomically set pred's infoNext's reference to new node.
     * no need to change pred's stamp.
     */


    curr->unlock();
    pred->unlock();
    return true;
    /*
     * unlock pred and curr.
     * return true.
     */
  }
    }
    curr->unlock();
    pred->unlock();
  }
}
```

A.1.5 The contains method

C++ Code.

```
/*
 * contains traverses the list from the head and
 * searches the list for the first node with key whose
 * value is greater than or equal to the required key "k".
 * returns true if key k is present in the list.
 * else, returns false.
 */
bool contains(int k)
{
  Node *pred, *curr, *succ;
```

```
    int predSt, currSt, nPredSt;
    int currKey;

    bool breakTest = false;

retry:
    while(true) {
      pred = head;
      /*
       * Start from the head.
       */

      curr = (pred->infoNext).get(&predSt);
      /*
       * Read node's infoNext's reference & stamp atomically.
       */

      currKey = curr->key;
      /*
       * read curr's key value.
       */

      while(true) {
        breakTest = (k <= currKey);
        /*
         * Break when key greater than or equal to required key is found.
         */

        succ = (curr->infoNext).get(&currSt);
        /*
         * atomically read curr's infoNext's reference(successor) and
    stamp.
         * Successor may be null if curr has been deleted.
         */

        nPredSt = (pred->infoNext).getStamp();
        /*
         * Read pred's stamp again before advancing forward.
         * This is the safety check to ensure we are traversing the list
         * correctly, in increasing order of keys.
         */

        if(predSt != nPredSt)
          goto retry;
          /*
```

```
        * If pred's new stamp is different from the one read previously,
        * a synchronization conflict is detected.
        * curr may have been deleted by another thread from the list.
        * The thread restarts it's traversal to ensure correctness.
        * If pred's stamp is still the same, then everything is fine.
        */

        if(breakTest)
          break;
        /*
        * If pred's stamp has not changed, everything is fine.
        * Check if required node has been found.
        * If yes, break. Else, continue.
        */

        pred = curr;
        curr = succ;
        predSt = currSt;
        /*
        * Keep advancing pred and curr in the list.
        * Keep track of new pred's old stamp to be used later,
        * to detect synchronization conflicts.
        */

        currKey = curr->key;
        /*
        * read curr's key value.
        */
    }

    if(currKey == k)
      return true;
    else
      return false;
      /*
    * if key is present in the list, return true.
    * else return false.
    */
  }
}
```

A.2 GCLFList Code

A.2.1 The find method

C++ Code.

```
/*
 * find method traverses the list from the head and
 * searches the list for the first node with key whose
 * value is greater than or equal to the required key "k".
 * returns a window object of the said node, node's predecessor
 * along with their stamps.
 */
Window* find(Node *head, int k, Node *prevCurr)
{
  Node *pred, *curr, *succ;
  int predSt, currSt, nPredSt;

  bool breakTest = false, snip = false;

retry:
  while (true) {
    pred = head;
    /*
     * Start from the head.
     */

    curr = (pred->infoNext).get(&predSt);
    /*
     * Read node's infoNext's reference & stamp atomically.
     */

    while (true) {
      int currKey = curr->key;
      /*
       * read curr's key value
       */

      succ = (curr->infoNext).get(&currSt);
      /*
       * atomically read curr's infoNext's reference(successor) and
    stamp.
       * Successor may be null if curr has been deleted.
       */

      if (currSt % 2 == 1) {
```

```
      snip = (pred->infoNext).compareAndSet(curr, succ, predSt,
         predSt+2);
   /*
   * This is the "helping" step.
   * if curr is marked(stamp is odd), attempt to physically remove
from the list.
   * done by calling an atomic CAS operation on pred.
   * to atomically set pred's infoNext's reference to successor and
increment stamp by 2.
   */
   if (!snip)
     goto retry;
     /*
   * If the CAS operation fails, restart the traversal.
   */

   Pool->set(curr);
   predSt += 2;
   /*
   * Else, add curr to the Pool.
   * And keep track of updated pred's stamp.
   */
}

breakTest = (k <= currKey);
/*
* Break when key greater than or equal to required key is found.
*/

nPredSt = (pred->infoNext).getStamp();
/*
* Read pred's stamp again before advancing forward.
* This is the safety check to ensure we are traversing the list
* correctly, in increasing order of keys.
*/

if (predSt != nPredSt)
  goto retry;
/*
* If pred's new stamp is different from the one read previously,
* a synchronization conflict is detected.
* curr may have been deleted by another thread from the list.
* The thread restarts it's traversal to ensure correctness.
* If pred's stamp is still the same, then everything is fine.
*/
```

```
      if (breakTest)
        break;
      /*
       * If pred's stamp has not changed, everything is fine.
       * Check if required pair of nodes has been found.
       * If yes, break. Else, continue.
       */

      if (!snip) {
        pred = curr;
        curr = succ;
        predSt = currSt;
        /*
         * If no helping was done i.e. no marked node was found,
         * keep advancing pred and curr in the list.
         * Keep track of new pred's old stamp to be used later,
         * to detect synchronization conflicts.
         */
      }
      else {
        curr = succ;
        snip = false;
        /*
         * If helping was done to remove an encountered marked done,
         * It implies pred is still the same. Advance only curr.
         */
      }
    }
    return new Window(pred, curr, predSt, currSt);
    /*
     * return pred and curr, along with their stamps.
     * encapsulated in a window object.
     */
    }
}
```

A.2.2 The remove method

C++ Code.

```
/*
 * remove a node with key value "k" from the list.
 * uses the find method to get the node with key value greater than or
   equal to k,
```

```
 * and it's predecessor.
 * removes the node, if present, from the list in two steps.
 * returns true only if key was present to be removed.
 * else returns false.
 */
bool remove(int k)
{
  bool snip;
  while (true) {
    Window *window = find(head, k, nullptr);
    /*
     * call find method to traverse list optimistically.
     * gets pred and curr, along with their stamps, in a window object
     */

    Node *pred = window->pred, *curr = window->curr;
    int predSt = window->predSt, currSt = window->currSt;
    /*
     * retrieve pred and curr, and their stamps, from the window object.
     */

    if (curr->key != k) {
      return false;
      /*
       * if key is not present, return false.
       */
    }
    else {
      Node *succ = (curr->infoNext).getReference();
      /*
       * read curr's infoNext's reference.
       */

      snip = (curr->infoNext).compareAndSet(succ, succ, currSt, currSt
        +1);
      /*
       * Deletion Step 1:
       * atomically increment curr's infoNext's stamp by 1 using CAS.
       * Logical Deletion.
       */

      if (!snip)
        continue;
      /*
       * If CAS fails, restart the operation.
```

```
      */

      if ((pred->infoNext).compareAndSet(curr, succ, predSt, predSt+2))
      {
        /*
        * Deletion Step 2:
        * atomically swing pred's infoNext's reference to successor.
        * and increment pred's infoNext's stamp by 2.
        * Physical Deletion.
        */

        Pool->set(curr);
        /*
        * If physical deletion is successful, add curr to the Pool.
        */
      }
      else {
        find(head, k, curr);
        /*
        * This step is optional.
        * If physical deletion is unsuccessful, retraverse the list to
    remove it.
        * Or depend on some other thread to "help out".
        */
      }
      return true;
      /*
      * return true on successful deletion.
      * Note: Will return true even if only Logical deletion is
    successful.
      */
    }
  }
}
```

A.2.3 The add method

C++ Code.

```
/*
* add a node with key value "k" to the list.
* uses the find method to get the node with key value greater than or
   equal to k,
* and it's predecessor.
* adds the node, if not already present, to the list in two steps.
```

```
 * returns true only if key was successfully added.
 * else returns false.
 */
bool add(int k)
{
  bool fromPool = false;
  Node *node;
  int nodeSt;

  node = Pool->get();
  /*
   * query the Pool for a node.
   */

  if (node == nullptr) {
    node = new Node(k);
    fromPool = false;
    /*
     * If Pool is empty, create a new node.
     */
  }
  else {
    node->key = k;
    nodeSt = (node->infoNext).getStamp();
    (node->infoNext).set(nullptr,nodeSt+1);
    fromPool = true;
    /*
     * else, node successfully retrieved from the Pool.
     * Increment new node's stamp by 1.
     * To make the stamp even.
     */
  }

  while (true) {
    Window *window = find(head, k, nullptr);
    /*
     * call find method to traverse list optimistically.
     * gets pred and curr, along with their stamps, in a window object
     */

    Node *pred = window->pred, *curr = window->curr;
    int predSt = window->predSt, currSt = window->currSt;
    /*
     * retrieve pred and curr, and their stamps, from the window object.
     */
```

```
 if (curr->key != k) {
    nodeSt = (node->infoNext).getStamp();
    (node->infoNext).set(curr,nodeSt);
    /*
    * If key is not already present in the list.
    * set new node's infoNext's reference to curr.
    */

    if ((pred->infoNext).compareAndSet(curr, node, predSt, predSt))
    {
      return true;
    }
    /*
    * attempt to atomically CAS pred's infoNext's reference to new
node
    * If CAS succeeds, return true.
    * Else, restart the operation.
    * Note: Next iteration, some other thread may have added the new
key instead.
    * Is so, then this thread will then return false.
    */
 }
 else {
 /*
 * key is already present in the list.
 */
    if (!fromPool) {
      delete node;
      /*
      * new node was newly created by this thread.
      * new node can be freed.
      * since no other thread has a reference to this node.
      */
    }
    else {
      nodeSt = (node->infoNext).getStamp();
      (node->infoNext).set(nullptr, nodeSt-1);
      Pool->set(node);
      /*
      * node was retrieved from the Pool.
      * decrement node's stamp to make it odd again.
      * add the node back to the Pool.
      */
    }
```

```
      return false;
      /*
       * return false since key already present.
       */
    }
  }
}
```

A.2.4   The contains method

C++ Code.

```
/*
 * contains traverses the list from the head and
 * searches the list for the first node with key whose
 * value is greater than or equal to the required key "k".
 * returns true if key k is present in the list.
 * else, returns false.
 */
bool contains(int k)
{
  Node *pred, *curr, *succ;
  int predSt, currSt, nPredSt;
  int currKey;

  bool breakTest = false;

  retry:
    while (true) {
      pred = head;
      /*
       * Start from the head.
       */

      while (true) {
        curr = (pred->infoNext).get(&predSt);
        /*
         * Read node's infoNext's reference & stamp atomically.
         */

        currKey = curr->key;
        /*
         * read curr's key value.
         */
```

```
    succ = (curr->infoNext).get(&currSt);
    /*
    * atomically read curr's infoNext's reference(successor) and
stamp.
    * Successor may be null if curr has been deleted.
    */

    breakTest = (k <= currKey);
    /*
    * Break when key greater than or equal to required key is found
.
    */

    nPredSt = (pred->infoNext).getStamp();
    /*
    * Read pred's stamp again before advancing forward.
    * This is the safety check to ensure we are traversing the list
    * correctly, in increasing order of keys.
    */

    if (predSt != nPredSt)
      goto retry;
      /*
    * If pred's new stamp is different from the one read previously
,
    * a synchronization conflict is detected.
    * curr may have been deleted by another thread from the list.
    * The thread restarts it's traversal to ensure correctness.
    * If pred's stamp is still the same, then everything is fine.
    */

    if (breakTest)
      break;
    /*
    * If pred's stamp has not changed, everything is fine.
    * Check if required node has been found.
    * If yes, break. Else, continue.
    */

    pred = curr;
    predSt = currSt;
    /*
    * Keep advancing pred in the list.
    * Keep track of new pred's old stamp to be used later,
    * to detect synchronization conflicts.
```

```
          */
        }
        bool marked = (currSt % 2 == 1);
        /*
         * check if curr is marked i.e. odd stamp.
         */

        return(currKey == k && !marked);
        /*
         * return true if and only if key is found and node is unmarked.
         * else, return false.
         */
    }
}
```

B   The Pool

B.1   Lock-Based Queue

```
class LBQueue
{
public:
  mutex enqLock,deqLock;
  Node *head,*tail;

  LBQueue() {
    head=new Node(0);
    tail=new Node(0);
    (tail->infoNext).set(nullptr,0);
    (head->infoNext).set(tail,0);
  }

  void set(Node *x)
  {
    enqLock.lock();
    if(x != nullptr) {
      int stamp = (tail->infoNext).getStamp();
      (tail->infoNext).set(x,stamp);
      tail=x;
    }
    enqLock.unlock();
  }

  Node* get()
```

```
  {
    Node* result = nullptr;
    deqLock.lock();
    if ((head->infoNext).getReference() != tail) {
      result = head;
      head = (head->infoNext).getReference();
    }
    deqLock.unlock();
    return result;
  }
};
```

B.2   Lock-Free Queue

```
class LFQueue {
public:
 class QNode;
 AtomicStampedReference<QNode> *head;
 AtomicStampedReference<QNode> *tail;

 class QNode {
  public:
  Node *node;
  AtomicStampedReference<QNode> next;

  QNode() {
   node = nullptr;
   next.set(nullptr, 0);
  }

  QNode(Node *node) {
   this->node = node;
   next.set(nullptr, 0);
  }
 };

 LFQueue() {
  QNode *sentinel = new QNode();
  head = new AtomicStampedReference<QNode>(sentinel, 0);
  tail = new AtomicStampedReference<QNode>(sentinel, 0);
 }

 void set(Node *node) {
  int lastStamp, nextStamp, stamp;

  if (node == nullptr)
```

```cpp
   return;
  QNode *x = new QNode(node);

  while (true) {
   QNode *last = tail->get(&lastStamp);
   QNode *next = (last->next).get(&nextStamp);

   if (last == tail->get(&stamp) && stamp == lastStamp) {
    if (next == nullptr) {
     if ((last->next).compareAndSet(next, x, nextStamp, nextStamp+1)) {
      tail->compareAndSet(last, x, lastStamp, lastStamp+1);
      return;
     }
    }
    else {
     tail->compareAndSet(last, next, lastStamp, lastStamp+1);
    }
   }
  }
}

Node* get() {
 int lastStamp, firstStamp, nextStamp, stamp;

 while (true) {
  QNode *first = head->get(&firstStamp);
  QNode *last = tail->get(&lastStamp);
  QNode *next = (first->next).get(&nextStamp);

  if (first == head->get(&stamp) && stamp == firstStamp) {
   if (first == last) {
    if (next == nullptr) {
     return nullptr;
    }
    tail->compareAndSet(last, next, lastStamp, lastStamp+1);
   }
   else {
    Node *ret = first->node;
    if (head->compareAndSet(first, next, firstStamp, firstStamp+1)) {
     return ret;
    }
   }
  }
 }
}
```

```
};
```