Non-Blocking Doubly-Linked Lists with Good Amortized Complexity

Niloufar Shafiei

Department of Electrical Engineering and Computer Science, York University, 4700 Keele Street, Toronto, Ontario, Canada
niloo@cse.yorku.ca

Abstract

We present a new non-blocking doubly-linked list implementation for an asynchronous shared-memory system. It is the first such implementation for which an upper bound on amortized time complexity has been proved. In our implementation, operations access the list via *cursors*. Each cursor is located at an item in the list and is local to a process. In our implementation, cursors can be used to traverse and update the list, even as concurrent operations modify the list. The implementation supports two update operations, insertBefore and delete, and two move operations, moveRight and moveLeft. An insertBefore(c, x) operation inserts an item x into the list immediately before the cursor c's location. A delete(c) operation removes the item at the cursor c's location and sets the cursor to the next item in the list. The move operations move the cursor one position to the right or left. Update operations use single-word Compare&Swap instructions. Move operations only read shared memory and never change the state of the data structure. If all update operations modify different parts of the list, they run completely concurrently. A cursor is active if it is initialized, but not yet removed from the process's set of cursors. Let $\dot{c}(op)$ be the maximum number of active cursors at any one time during the operation op. The amortized step complexity is $O(\dot{c}(op))$ for each update op and $O(1)$ for each move. We provide a detailed correctness proof and amortized analysis of our implementation.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases non-blocking data structure, doubly-linked list, shared memory, amortized complexity, cursor

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.35

1 Introduction

The linked list is a fundamental data structure that has many applications in distributed systems including processor scheduling, memory management and sparse matrix computations [9, 15, 17]. It is also used as a building block for more complicated data structures such as deques, skip graphs and Fibonacci heaps. We design a concurrent doubly-linked list for asynchronous shared-memory systems that is *non-blocking* (also sometimes called *lock-free*): it guarantees some operation will complete in a finite number of steps. Our list has a full proof of correctness and analysis of its amortized complexity. The first non-blocking *singly*-linked list [23] was proposed two decades ago. Designing a non-blocking *doubly*-linked list was an open problem for a long time. Doubly-linked lists have been implemented using multi-word synchronization primitives that are not widely available [1, 10]. Sundell and Tsigas [21] gave the first implementation from single-word compare&swap (CAS). However, they did not provide a correctness proof and their implementation has some problems. (See Section 2.)

A process accesses our list via a *cursor*, which is an object in the process's local memory that is located at an item in the list. Update operations can insert or delete an item at the cursor's location, and moveLeft and moveRight operations move the cursor to the adjacent item in either direction. If the item where a cursor c located at is removed by another process, an operation called with c first needs to *recover c's location* in the list. In our list, recovering a cursor's location and moving a cursor are achieved using only reads of shared memory, even when there are concurrent updates. Thus, in our list, only updates might interfere with other concurrent operations. However, if all concurrent updates are on disjoint parts of the list, they do not interfere with one another. Our implementation is modular and can be adapted for other updates, such as replacing one item by another. For simplicity, we assume the existence of a garbage collector such as the one in Java.

In Section 3, we give a novel specification that describes how updates affect cursors and how a process gets feedback about other processes' updates at the location of its cursor. This interface makes the list easy to use as a black box. In our implementation, a cursor c becomes *invalid* if an update is performed at c's location using another cursor. If an operation is called with an invalid cursor, it returns invalidCursor and makes the cursor valid again. This avoids having a process perform an operation on the wrong item. If an insertion is performed before a cursor c using another cursor, c becomes invalid for insertions only, to ensure that an item can be inserted between two specific items. This makes it easy to maintain a sorted list. For example, if two processes try to insert 5 and 7 at the same location simultaneously, one fails and returns invalidCursor. This avoids inserting 7 and then 5 out of order.

We provide a detailed proof that our list is *linearizable*: each operation appears to take place atomically at some time during the operation. One of the main challenges is to ensure the two pointer changes required by an update appear to occur atomically. Our implementation uses two CAS steps to change the pointers. Each update appears to take effect at the first CAS. Between the two CAS steps, the data structure is temporarily inconsistent. We design a mechanism for detecting such inconsistencies by only reading the shared memory, which allows concurrent operations to behave as if the second change has already occurred. This makes moves and recovering cursors' locations very efficient.

We give an amortized analysis of our implementation (excluding garbage collection), which is the first for any non-blocking doubly-linked list. A cursor is active if it is initialized, but not removed from the process's set of cursors. Let $\dot{c}(op)$ be the maximum number of active cursors at any one time during the operation op. The amortized complexity of each operation op is $O(\dot{c}(op))$ for updates and $O(1)$ for moves. Due to space restrictions, we sketch the proof of correctness and amortized analysis here. Complete details are in [19].

To summarize the contributions of this paper:

- We present a non-blocking linearizable doubly-linked list using single-word CAS.
- The cursors provided by our implementation are robust: they can be used to traverse and update the list, even as concurrent operations modify the list.
- Cursors' locations are recovered and cursors are moved by only reading the shared memory.
- Our implementation and proof are modular and can be adapted for other data structures.
- Our implementation can easily maintain a sorted list.
- In our list, the amortized complexity of each update op is $O(\dot{c}(op))$ and each move is $O(1)$.
- We empirically show our list outperforms the one in [21] on a multi-core machine.

2 Related Work

In this paper, we focus on non-blocking algorithms of doubly-linked lists, which do not use locks. There are two general techniques for obtaining non-blocking data structures: universal constructions (see [7] for a survey) and transactional memory (see [11] for a survey). Such

Implementation	supports cursors?	operations to move cursor?	recover cursor's location?	primitive used	# of CAS with no contention
Greenwald [10]	No	-	-	2-CAS	depends on size of list
Attiya and Hillel [1]	Yes	No	No	2-CAS	13-15
Sundell and Tsigas [21]	Yes	Yes (CAS used)	Yes (CAS used)	CAS	2-4
List presented here	Yes	Yes (No CAS)	Yes (No CAS)	CAS	5

Figure 1 Implementations of doubly-linked lists.

general techniques are usually less efficient than implementations designed for specific data structures. Turek, Shasha and Prakash [22] and Barnes [2] introduced a technique in which processes cooperate to complete operations to ensure non-blocking progress. Each update operation stores information that other processes can use to help complete the update in a descriptor object. This technique has been used for various data structures. Here, we extend the scheme used in [3, 4, 5, 8] to coordinate processes for tree structures and the scheme used in [18] for updates that make more than one change to a Patricia trie. The implementations in [3, 4, 5, 8] only handle one change atomically, but updates in our list make multiple changes atomically using a simpler scheme than the one used in [18]. However, handling the cursors and move operations in our list are original.

The k-CAS primitive modifies k locations atomically. Although it is usually not available in hardware, it can be implemented from single-word CAS [12, 16, 20]. Doubly-linked lists can be implemented using k-CAS, but it is not completely straightforward to do so. Suppose each item is represented by a node with *nxt* and *prv* fields that point to the adjacent nodes. Consider a list of four nodes, A, B, C and D. A deletion of C must change the *nxt* pointer in B from C to D and the *prv* pointer in D from C to B. It is *not* sufficient for the deletion to update these two pointers with a 2-CAS. If two concurrent deletions remove B and C in this way, C would still be accessible through A after the deletions. This problem can be avoided if the deletion of C uses a 4-CAS to simultaneously update the two pointers and check whether the two pointers of C still point to B and D. Then, the 4-CAS of one of the two concurrent deletions would fail. The most efficient k-CAS implementation [20] uses $2k + 1$ CAS steps to update k words when there is no contention. Thus, at least 9 CAS steps would be used for a 4-CAS. Moreover, although the 4-CAS works for updating pointers, it is not obvious how to recover a cursor's location when another process deletes the cursor's node.

Greenwald's doubly-linked list [10] uses 2-CAS, but does not provide cursors. His approach does not support concurrency: all processes cooperate to execute one operation at a time.

Attiya and Hillel [1] proposed a doubly-linked list using 2-CAS, but it only supports update operations. It has the nice property that concurrent updates can interfere with one another only if they are changing nodes close to each other. If there is no interference, an update performs 13 to 15 CAS steps (and one 2-CAS). Their implementation does not recover a cursor's location, so deletions might make other processes lose their place in the list. They also give a restricted implementation using single-word CAS, in which deletions can be performed only at the ends of the list.

None of the implementations that use k-CAS handle cursors with the same functionality as ours. (See Figure 1.) Since the implementations in [1, 10] do not provide a way to traverse the list, they are not complete implementations of a doubly-linked list. Moreover, they all perform many CAS steps for contention-free updates, whereas ours performs only five.

Sundell and Tsigas [21] gave the first non-blocking doubly-linked list using single-word CAS (although a word must store both a bit and a pointer). Non-blocking data structures are notoriously difficult to design, so detailed correctness proofs are essential. In [21], the claim of linearizability is justified by defining linearization points without proving that they are correct. The implementation has at least minor errors: using the Java PathFinder model checker [13], we found an execution that incorrectly dereferences a null pointer. We contacted the authors, who suggested changing some lines to fix the problem, but a correctness proof of the revised version is still lacking. Their implementation is ingenious but quite complicated. In particular, the helping mechanism is very complex, partly because updates can terminate before completing the necessary changes to the list, so operations may have to help non-concurrent updates.

There are a number of differences between the designs of our list and the one in [21]. In [21], to recover a cursor's location or to move a cursor, sometimes CAS steps are performed. Our approach is quite different, allowing a cursor's location to be recovered and cursors to be moved only by reading shared memory, even when there is contention. So, moves do not interfere with one another. This is a desirable property since moves are more common than updates in many applications. In the best case, the updates in [21] perform 2 to 4 CAS steps. However, moves must perform CAS steps to help complete updates. In fact, deletions can construct long chains of deleted nodes whose pointers do not get updated by the deletions. Then, a move may have to traverse this chain, performing CAS steps at every node. In our list, an update helps only updates that are concurrent with itself, and moves do not help at all. Our implementation can easily be used to maintain a sorted list. It is not straightforward to see how the implementation in [21] could maintain a sorted list. Our empirical evaluation shows our list outperforms the one in [21] on a multi-core machine.

3 The Sequential Specification

We give a sequential specification, which describes how operations behave when performed one at a time. (A more formal specification is available in [19].) This specification is extended to concurrent implementations by requiring them to be linearizable [14].

A list is a pair (L, S) where L is a finite sequence of items ending with a special end-of-list marker (EOL), and S is a set of cursors. Each cursor c in S is located at one item in L and $c.item$ presents that item. Eight types of operations are supported: initializeCursor, destroyCursor, resetCursor, moveRight, moveLeft, get, delete and insertBefore. An **initializeCursor**(c) adds new cursor c to S whose item is the first item in L. A **destroyCursor**(c) removes cursor c from S. A process p can call an operation with a cursor c only if p itself initialized c and c has not been removed from S. A **resetCursor**(c) locates c at the first item in L. A **moveRight**(c) advances c's location to the next item in L and returns true, (unless it is already at the last item in L, in which case it returns false and has no effect). Similarly, **moveLeft**(c) sets c's location to the previous item in L and returns true, (unless it is already at the first item in L, in which case it returns false and has no effect). A **get**(c) returns the value of $c.item$.

To keep track of cursor invalidation, each cursor in S has two additional fields called $invDel$ and $invIns$, which indicates whether the cursor is invalid for different operations. A **delete**(c) removes the item that c is located at and returns true, (unless c is located at EOL in which case the delete returns false). If the deletion is successful, it also moves each cursor c' located at the deleted item to the next item in L. This ensures that all cursors continue to point to items that are currently in the list, so that no other process can lose its place in the list as a result of the deletion. For each cursor $c' \neq c$ that is moved as a result of the delete, $c'.invDel$ is set to true so that if the next operation called with c' is an update, get or move, it returns invalidCursor to indicate that c' has been moved. This ensures that the operation is not inadvertently applied to the wrong location. For example, suppose a process p performs a delete(c) that removes item x. If another process p' has cursor c' located at x, $c'.item$ is set to the next item y in L and c' becomes invalid (i.e., $c'.invDel$ becomes true). Thus, the deletion cannot cause c' to lose its place in L. Since x is removed by p, p' does not yet know that c' is no longer located at x. If p' then calls a delete(c') to attempt to remove x, since $c'.invDel$ is true, it returns invalidCursor and does not remove y. When $c'.invDel$ is true, the next operation called with c' sets $c'.invDel$ to false, making c' valid again.

An **insertBefore**(c, v) adds a new item with value v to the left of c's location and returns true. When an insertBefore(c, v) succeeds, each other cursor c' located at the same item as c becomes invalid for insertions only. This is indicated by setting $c'.invIns$ to true. If an insertBefore(c', v') operation is called when $c'.invIns$ is true, it returns invalidCursor. This invalidation ensures an item can be inserted between two specific items in the list. For example, suppose we wish to maintain L so that values of items are sorted and process p' has a cursor c' whose item's value is 5. Then, p' advances c' to the next item in the sequence, which has value 8. If 7 is inserted before 8 by another process p, c' becomes invalid only for insertions (i.e., $c'.invIns$ becomes true). Since 7 is inserted by p, p' does not yet know that the item before 8 is 7. If p' then calls an insertBefore operation with c' to attempt to insert 6 before 8, it does not succeed because that would place 6 between 7 and 8. Since $c'.invIns$ is true, insertion by p' returns invalidCursor instead. When $c'.invIns$ is true, the next operation called with c' sets $c'.invIns$ to false, making c' valid for insertions again.

4 The Non-blocking Implementation

List items are represented by Node objects, which have pointers to adjacent Nodes. A Cursor object is simply a pointer to a Node in a process's local memory. Updates are done in several steps as shown in Fig. 2 and 3. To avoid simultaneous updates to overlapping parts of the list, an update flags a Node before removing it or changing one of its pointers. This flag acts like a lock on the Node's pointers. To ensure the non-blocking property, other operations can help complete the update that placed the flag and then remove the flag. To facilitate this, a Node is flagged by storing a pointer to an Info object, which is a descriptor of the update and contains the information needed to help complete the update. List pointers are updated using CAS, so that helpers cannot perform an operation more than once.

The correctness of algorithms using CAS often depends on the fact that, if a CAS on variable V succeeds, V has not changed since an earlier read. An ABA problem occurs when V changes from one value to another and back before the CAS occurs, causing the CAS to succeed when it should not. When a Node new is inserted between Node x and y, we replace y by a new copy, $yCopy$ (Fig. 3). This avoids an ABA problem that would occur if, instead, insertBefore simply changed the pointers in x and y to new, because a subsequent deletion of new could then change x's pointer back to y again. Creating a new copy of y also makes invalidation of Cursors for insertions easy. An insertion of a Node before y writes a permanent pointer in y to $yCopy$ before replacing y, so that any other process whose Cursor is at y can detect that an insertion has occurred there and update its Cursor to $yCopy$.

The objects used in our implementation are described in line 1 to 16 of Fig. 4. A Node has the following fields. The val field contains the item's value, nxt and prv point to the next and previous Nodes in the list, $copy$ points to a new copy of the Node (if any), $info$ points to an Info object that is the descriptor of the update that last flagged the Node, and $state$ is initially ordinary and is set to copied (before the Node is replaced by a new copy)

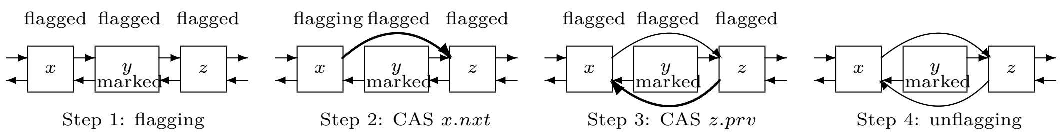

Figure 2 A delete operation.

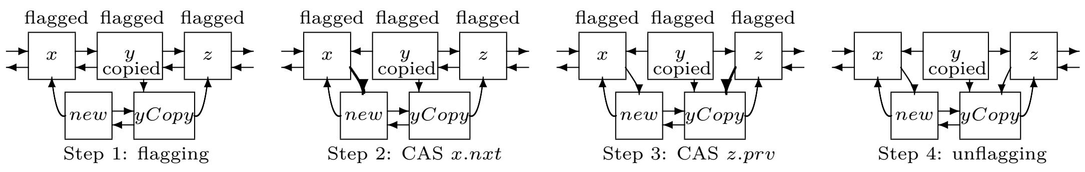

Figure 3 An insertBefore operation.

or marked (before the Node is removed). The *info* field is initially set to a dummy Info object, *dum*. The *info*, *nxt* and *prv* fields of a Node are changed using CAS steps. We call the steps that try to modify these three fields *flag CAS*, *forward CAS* and *backward CAS* steps, respectively. To avoid special cases, we add sentinel Nodes *head* and *tail*, which do not contain values, at the ends of the list. They are never changed and Cursors never move to *head* or *tail*. The last Node before *tail* always contains the value EOL.

Info objects are used as our update operation descriptors. An Info object I has the following fields. $I.nodes[0..2]$ stores the three Nodes x, y, z to be flagged before changing the list. $I.oldInfo[0..2]$ stores the expected values to be used by the flag CAS steps on x, y and z. $I.newNxt$ and $I.newPrv$ store the new values for the forward and backward CAS steps on $x.nxt$ and $z.prv$. $I.rmv$ indicates whether y should be removed from the list or replaced by a new copy. $I.status$, indicates whether the update is inProgress (the initial value), committed (after the update is completed) or aborted (after a Node is not flagged successfully). (One exception is the dummy Info object *dum* whose *status* is initially aborted.) The *status* field is the only field of I whose value might be changed after I's creation. A Node is *flagged for* I if its *info* field is I and $I.status =$ inProgress. Thus, setting $I.status$ to committed or aborted also has the effect of removing I's flags. As with locks, successful flagging of the three Nodes guarantees that the operation will be completed successfully without interference from other operations. Unlike locks, if the process performing an update crashes after flagging, other processes may complete its update using the information in I.

4.1 Detailed Description of the Algorithms

Pseudo-code for our implementation is given in Fig. 4.

We say a Node v is *reachable* if there is a path of *nxt* pointers from *head* to v. At all times, the reachable Nodes correspond to the items in the list. So, each update is linearized when its forward CAS succeeds (step 2 of Fig. 2 and 3). Just after this CAS, y becomes unreachable. We prove that no process changes $y.nxt$ or $y.prv$ after that, so $y.prv$ remains equal to x. Since there is no ABA problem, $x.nxt$ is never set back to y after y becomes unreachable. Thus, the test $y.prv.nxt \neq y$ tells us whether y has become unreachable (even between the successful forward and backward CAS of the update that removed y). This test is used in recovering the location of a Cursor whose *node* is removed from the list (line 77) and also by moveLeft operations (line 47).

Since a Cursor c is a pointer in a process's local memory, it becomes out of date if the Node it points to is deleted or replaced by another process's update. Thus, at the beginning of

```
type Cursor
  Node node                                   ▷ location of Cursor

type Node                                     ▷ represent list item
  Value val
  Node nxt                                    ▷ next Node
  Node prv                                    ▷ previous Node
  Node copy                   ▷ new copy of Node (if replaced)
  Info info                                   ▷ descriptor of update
  {copied, marked, ordinary} state ▷ indicates if Node
    is copied by an insertion or marked for deletion

type Info                          ▷ Descriptor of an update
  Node[3] nodes                               ▷ Nodes to be flagged
  Info[3] oldInfo ▷ expected values of CASs that flag
  Node newNxt                   ▷ set $nodes[0].nxt$ to this
  Node newPrv                   ▷ set $nodes[2].prv$ to this
  Boolean rmv                ▷ is $I.nodes[1]$ being deleted?
  {inProgress, committed, aborted} status

insertBefore($c$: Cursor, $v$: Value):{true, invalidCursor}
  while (true){
    $\langle y, yInfo, z, x, invDel, invIns \rangle \leftarrow$
      updateCursor($c$)              ▷ recover $c$'s location
    if $invDel$ or $invIns$ then return invalidCursor
    $nodes \leftarrow [x, y, z]$
    $oldI \leftarrow [x.info, yInfo, z.info]$
    if checkInfo($nodes$, $oldI$) then{ ▷ no interference
      $new \leftarrow$ new Node($v$, null, $x$, null, $dum$, ordinary)
      $yCopy \leftarrow$ new Node($y.val$, $z$, $new$, null, $dum$,
        ordinary)
      $new.nxt \leftarrow yCopy$
      $I \leftarrow$ new Info($nodes$, $oldI$, $new$, $yCopy$, false,
        inProgress)                   ▷ create descriptor
      if help($I$) then{              ▷ if insert completed
        $c.node \leftarrow yCopy$          ▷ move $c$ to new copy
        return true }}}

delete($c$: Cursor):{true, false, invalidCursor}
  while (true){
    $\langle y, yInfo, z, x, invDel, \text{-} \rangle \leftarrow$ updateCursor($c$)
                                     ▷ recover $c$'s location
    if $invDel$ then return invalidCursor
    $nodes \leftarrow [x, y, z]$
    $oldI \leftarrow [x.info, yInfo, z.info]$
    if checkInfo($nodes$, $oldI$) then{ ▷ no interference
      if $y.val$ = EOL then return false ▷ $c$ is at last item
      $I \leftarrow$ new Info($nodes$, $oldI$, $z$, $x$, true, inProgress)
                                     ▷ create descriptor
      if help($I$) then{              ▷ if delete completed
        $c.node \leftarrow z$              ▷ move $c$ to next item
        return true}}}

moveLeft($c$: Cursor):{true, false, invalidCursor}
  $\langle y, -, -, x, invDel, \text{-} \rangle \leftarrow$ updateCursor($c$)
                                     ▷ recover $c$'s location
  if $invDel$ then return invalidCursor
  if $x = head$ then return false     ▷ $y$ is the 1st item
  if $x.prv.nxt \neq x$ and $x.nxt = y$ then{ ▷ $x$ not in list
    if $x.state$ = copied then          ▷ $x$ is replaced
      $c.node \leftarrow x.copy$           ▷ move $c$ to new copy
    else{                               ▷ $x$ is deleted
      $w \leftarrow x.prv$              ▷ read the item before $x$
      if $w = head$ then return false ▷ $x$ was the 1st item
      $c.node \leftarrow w$}}         ▷ move $c$ to the item before $x$
  else $c.node \leftarrow x$          ▷ move $c$ to the item before $y$
  return true

moveRight($c$: Cursor):{true, false, invalidCursor}
  $\langle y, -, z, -, invDel, \text{-} \rangle \leftarrow$ updateCursor($c$)
                                     ▷ recover $c$'s location
  if $invDel$ then return invalidCursor
  if $y.val$ = EOL then return false  ▷ $y$ is the last item
  $c.node \leftarrow z$              ▷ move $c$ to the item after $y$
  return true

initializeCursor($c$: Cursor):ack
  $c.node \leftarrow head.nxt$
  return ack

destroyCursor($c$: Cursor):ack
  return ack

resetCursor($c$: Cursor):ack
  $c.node \leftarrow head.nxt$           ▷ move $c$ to the 1st item
  return ack

get($c$: Cursor):{Value, invalidCursor}
  $\langle y, -, -, -, invDel, \text{-} \rangle \leftarrow$ updateCursor($c$)
  if $invDel$ then return invalidCursor
  return $y.val$

updateCursor($c$: Cursor):⟨Node, Info, Node, Node,
    Boolean, Boolean⟩
  $invDel \leftarrow$ false
  $invIns \leftarrow$ false
  while( $c.node.prv.nxt \neq c.node$){ ▷ $c.node$ not in list
    if $c.node.state$ = copied then{     ▷ $c.node$ replaced
      $invIns \leftarrow$ true        ▷ make $c$ invalid for insertion
      $c.node \leftarrow c.node.copy$}       ▷ move $c$ to new copy
    if $c.node.state$ = marked then{      ▷ $c.node$ deleted
      $invDel \leftarrow$ true        ▷ make $c$ invalid for deletion
      $c.node \leftarrow c.node.nxt$}}   ▷ move $c$ to the next item
  $info \leftarrow c.node.info$         ▷ read info before pointers
  return $\langle c.node, info, c.node.nxt, c.node.prv, invDel,$
    $invIns \rangle$

checkInfo($nodes$: Node[3], $oldInfo$: Info[3]):Boolean
  for $i \leftarrow 0$ to 2{         ▷ detect other updates in progress
    if $oldInfo[i].status$ = inProgress then{
      help($oldInfo[i]$)                ▷ help other update
      return false}}                     ▷ retry my update
  for $i \leftarrow 0$ to 2               ▷ detect removed nodes
    if $nodes[i].state \neq$ ordinary then return false
  for $i \leftarrow 1$ to 2         ▷ if flag of 2nd or 3rd node fail
    if $nodes[i].info \neq oldInfo[i]$ then return false
  return true                     ▷ no interference detected

help($I$: Info):Boolean
  $doPtrCAS \leftarrow$ true, $i \leftarrow 0$
  while ($i < 3$ and $doPtrCAS$){
    CAS($I.nodes[i].info$, $I.oldInfo[i]$, $I$) ▷ flag CAS
    $doPtrCAS \leftarrow (I.nodes[i].info = I)$
    $i \leftarrow i + 1$}
  if $doPtrCAS$ then{                 ▷ flag CASs succeeded
    if $I.rmv$ then $I.nodes[1].state \leftarrow$ marked
    else{                              ▷ in case of insertion
      $I.nodes[1].copy \leftarrow I.newPrv$     ▷ set new copy
      $I.nodes[1].state \leftarrow$ copied}
    CAS($I.nodes[0].nxt$, $I.nodes[1]$, $I.newNxt$)
                                          ▷ forward CAS
    CAS($I.nodes[2].prv$, $I.nodes[1]$, $I.newPrv$)
                                         ▷ backward CAS
    $I.status \leftarrow$ committed} ▷ unflag of successful update
  else if $I.status$ = inProgress then $I.status \leftarrow$ aborted
                  ▷ unflag nodes for unsuccessful update
  return ($I.status$ = committed)
```

Figure 4 Pseudo-code for a non-blocking doubly-linked list.

an update, move or get operation called using c, **updateCursor**(c) is called to bring *c.node* up to date. If *c.node* has been replaced with a new copy by an insertBefore, updateCursor sets *invIns* to true (line 79) and follows the *copy* pointer (line 80). Similarly, if *c.node* has been deleted, updateCursor sets *invDel* to true (line 82) and follows the *nxt* pointer (line 83), which was the next Node at the time of deletion. UpdateCursor repeats the loop at line 77–83 until the test on line 77 indicates that *c.node* is in the list. At the end, updateCursor returns *c.node*, its *info*, *nxt* and *prv* field, *invDel* and *invIns* (line 85).

After calling updateCursor, each update *op* calls **checkInfo** to see if any Node that *op* wants to flag is flagged with an Info object I' of another update. If so, checkInfo calls help(I') (line 89) to try completing the other update, and returns false to indicate *op* should retry. Similarly, if checkInfo sees that another operation has already removed one of the Nodes (line 92) or changed the *info* field of y or z (line 94), it returns false, causing *op* to retry. If checkInfo returns true, *op* creates a new Info object I that describes the update *op* (line 27 or 39) and calls help(I) to try to complete the update (line 28 or 40).

The **help**(I) routine performs the real work of the update. First, it uses flag CAS steps to store I in the *info* fields of the Nodes to be flagged (line 99). If help(I) sees a Node v is not flagged successfully (line 100), help(I) checks if *I.status* is inProgress (line 110). If so, it follows that no helper of I succeeded in flagging all three Nodes; otherwise I's flag on v could not have been removed while I is inProgress. So, v was flagged by a different update before help(I)'s flag CAS. Thus, *I.status* is set to aborted (line 110) and help(I) returns false (line 111), causing *op* to retry.

If the Nodes x, y and z in *I.nodes* are all flagged successfully with I, *y.state* is set to marked (line 103) for a deletion, or copied (line 106) for an insertion. In the latter case, *y.copy* is first set to the new copy (line 105). Then, a forward CAS (line 107) changes *x.nxt* and a backward CAS (line 108) changes *z.prv*. Finally, help(I) sets *I.status* to committed (line 109) and returns true (line 111). A *CAS of I* refers to a CAS step executed inside help(I). We prove below that the *first* forward and *first* backward CAS of I among all calls to help(I) succeed (and no others do).

Both the **insertBefore**(c, v) and **delete**(c) operations have the same structure. They first call updateCursor(c) to bring the Cursor c up to date, and return invalidCursor if this routine indicates c has been invalidated. Then, they call checkInfo to see if there is interference by other updates. If not, they create an Info object I and call help(I) to complete the update. If unsuccessful, they retry.

A **moveRight**(c) calls updateCursor(c) (line 57), which sets *c.node* to a Node y and also returns a Node z read from *y.nxt*. We show there is a time during the move when y is reachable and $y.nxt = z$. If $y.val =$ EOL, the operation returns false (line 59). Otherwise, it sets *c.node* to z (line 60).

A **moveLeft**(c) is more complex because *prv* pointers are updated *after* an update's linearization point, so they are sometimes inconsistent with the true state of the list. A moveLeft first calls updateCursor(c) (line 44), which updates *c.node* to some Node y and also returns a Node x read from *y.prv*. If x is *head*, the operation cannot move c to *head* and returns false (line 46). If the test on line 47 indicates x is reachable, *c.node* is set to x (line 54). This is also done if $x.nxt \neq y$; in this case, we can show that y became unreachable during the move, but *x.nxt* pointed to y just before y became unreachable. Otherwise, x has become unreachable and the test $x.nxt = y$ on line 47 ensures that x was the element before y when it became unreachable. If x was replaced by an insertion, *c.node* is set to that replacement Node (line 49). If x was removed by a deletion, we set *c.node* to *x.prv* (line 53), unless that Node is *head*. We prove in Lemma 12, below, that whenever moveLeft updates *c.node* to some value v, there is a time during the operation when v is reachable and $v.nxt = y$.

5 Correctness Proof

The detailed proof of correctness (available in [19]) is about 50 pages long, so we give only a brief sketch. An execution is a sequence of configurations, $C_0, C_1, \ldots$ such that, for each $i \geq 0$, C_{i+1} follows from C_i by a step of the implementation. For the proof, we assign each Node v a positive real value, called its *abstract value*, denoted $v.absVal$. The $absVal$ of *head*, EOL and *tail* are 0, 1 and 2 respectively. When insertBefore creates the Nodes *new* and *yCopy* (see Fig. 3), $yCopy.absVal = y.absVal$ and $new.absVal = (x.absVal + y.absVal)/2$. The following basic invariant is straightforward to prove.

▶ **Invariant 1.**
- *Any field that is read in the pseudo-code is non-null.*
- *Cursors do not point to* head *or* tail.
- *If $v.nxt = tail$, then $v.val$ = EOL.*
- *If $v.nxt = w$ or $w.prv = v$, then $v.absVal < w.absVal$.*

5.1 Part 1: Flagging

Part 1 proves v is flagged for an Info object I when the first forward CAS or first backward CAS of I is applied to Node v. We first show there is no ABA problem on *info* fields.

▶ **Lemma 2.** *The* info *field is never set to a value that has been stored there previously.*

Proof Sketch. The old value used for I's flag CAS on Node v was read from $v.info$ before I is created. So, every time $v.info$ is changed from I' to I, I is a newer Info object than I'. ◀

By Lemma 2, only the first flag CAS of I on each Node in $I.nodes$ can succeed since all such CAS steps use the same expected value. We say I is *successful* if these three first flag CAS steps all succeed.

▶ **Lemma 3.** *After* v.info *is set to I, it remains equal to I until $I.status \neq$ inProgress.*

Proof Sketch. If $v.info$ is changed from I to I', the operation that created I' at line 27 or 39 first called checkInfo on line 23 or 37 and that call returned true. Thus, that call to checkInfo saw $I.status \neq$ inProgress at line 88. ◀

▶ **Observation 4.** *If a process executes line 103–109 inside help(I), I is already successful.*

▶ **Lemma 5.** *If I is successful, $I.status$ is never aborted. Else, $I.status$ is never committed.*

Proof Sketch. If I is not successful, the claim follows from Observation 4. If I is successful, the first flag CAS on each Node in $I.nodes$ succeeds. By Lemma 3, $doPtrCAS$ is not set to false on line 100 until $I.status \neq$ inProgress. So, every call to help(I) evaluates the test on line 102 to true until $I.status \neq$ inProgress. So, no process reaches line 110 before $I.status$ is set to committed on line 109. ◀

▶ **Lemma 6.** *For each of lines 103–109, when the first execution of that line among all calls to help(I) occurs, all Nodes in* I.nodes *are flagged for I.*

Proof Sketch. Suppose one of lines 103–109 is executed inside help(I). By Observation 4, a flag CAS of I already succeeded on each Node in $I.nodes$. By Lemma 5, $I.status$ is never aborted. By Lemma 3, all three Nodes remain flagged for I until some help(I) sets $I.status$ to committed on line 109. ◀

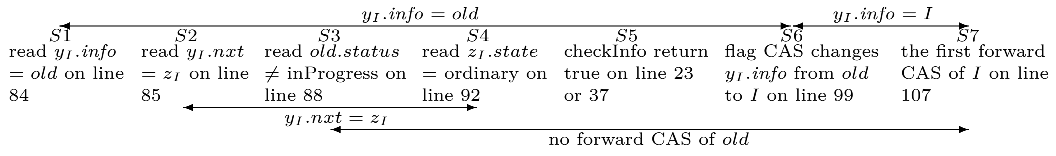
$y_I.info = old$
$y_I.info = I$
$S1$ read $y_I.info$ $= old$ on line 84
$S2$ read $y_I.nxt$ $= z_I$ on line 85
$S3$ read $old.status$ $\neq$ inProgress on line 88
$S4$ read $z_I.state$ $=$ ordinary on line 92
$S5$ checkInfo return true on line 23 or 37
$S6$ flag CAS changes $y_I.info$ from old to I on line 99
$S7$ the first forward CAS of I on line 107
$y_I.nxt = z_I$
no forward CAS of old

Figure 5 Sequence of events used in proof of Lemma 8, Statement 3.

5.2 Part 2: Forward and Backward CAS Steps

Let $\langle y_I, -, z_I, x_I, -, - \rangle$ be the result updateCursor(c) returns on line 19 or 33 before creating I on line 27 or 39. Part 2 of our proof shows that successful flagging ensures that x_I, y_I and z_I are three consecutive Nodes in the list just before the first forward CAS of I, and that the first forward and the first backward CAS of I succeed (and no others do).

▶ **Lemma 7.** *At all configurations after I becomes successful, y_I.info $= I$.*

Proof sketch. To derive a contradiction, assume $y_I.info$ is changed from I to I'. Before creating I', the call to checkInfo returns true, so it sees $I.status \neq$ inProgress at line 88 and then $y_I.state =$ ordinary at line 92. This contradicts the fact that before $I.status$ is set to committed at line 109, $y_I.state$ is set to a non-ordinary value at line 103 or 106 (and no step of the code can change it back to ordinary). ◀

▶ **Lemma 8.**
1. *The first forward and the first backward CAS of I succeed and all other forward and backward CAS steps of I fail.*
2. *The nxt or prv field of a Node is never set to a Node that has been stored there before.*
3. *At the configuration C before the first forward CAS of I, x_I, y_I and z_I are reachable, $x_I.nxt = y_I$, $y_I.prv = x_I$, $y_I.nxt = z_I$ and $z_I.prv = y_I$.*
4. *At all configurations after the first forward CAS of I, $y_I.prv = x_I$ and $y_I.nxt = z_I$.*

Proof sketch. We use induction on the length of the execution.
Statement 1: By induction hypothesis 3, the first forward CAS of I succeeds, since $x_I.nxt = y_I$ just before it. By induction hypothesis 2, no other forward CAS of I succeeds. By induction hypothesis 3, $z_I.prv$ was y_I at some time before the first backward CAS of I. All backward CASes of I use y_I as the expected value of $z_I.prv$, so only the first can succeed (by induction hypothesis 2). By Lemma 6, $z_I.info = I$ at the first forward and first backward CAS of I, and hence at all times between, by Lemma 2. By Lemma 6, no backward CAS of any other Info object changes $z_I.prv$ during this time. So, the first backward CAS of I succeeds.

Statement 2: Intuitively, when the nxt field changes from v to another value, v is thrown away and never used again. (See Fig. 2 and 3). Suppose the first forward CAS of I sets $x_I.nxt$. If I is created by an insertBefore, the CAS sets $x_I.nxt$ to a newly created Node. If I is created by a delete, $z_I.info = I$ at the first forward CAS of I, by Lemma 6. No forward CAS of another Info object I' can change $x_I.nxt$ from z_I to another value earlier, since then $z_I.info$ would have to be I' at the first forward CAS of I, by Lemma 7. The proof for prv fields is symmetric.

Statement 3: First, we prove $y_I.nxt = z_I$ at C. Before I can be created, the sequence of steps $S1, \ldots, S5$ shown in Fig. 5 must occur. By Lemma 6, $y_I.info$ is set to I by some step $S6$ and $y_I.info = I$ at $S7$. By Lemma 2, $y_I.info = old$ between $S1$ and $S6$ and $y_I.info = I$ between $S6$ and $S7$. So, by Lemma 6, only the first forward CAS of old can change $y_I.nxt$

between $S1$ and $S7$. Before $y_I.nxt$ can be changed from z_I to another value by help(old), $z_I.state$ is set to marked or copied (and it can never be changed back to ordinary). So, $y_I.nxt$ is still z_I at $S4$. The first forward CAS of old does not occur after $S3$ since $old.state$ is already committed or aborted at $S3$. So, $y_I.nxt$ is still z_I at C.

By a similar argument, $y_I.prv = x_I$ and x_I, y_I and z_I are reachable in C. The prv and nxt field of two adjacent reachable Nodes might not be consistent at C only if C is between the first forward and first backward CAS of some Info object I' and one of the two Nodes is flagged for I' (step 2 of Fig. 2 and 3). Since x_I, y_I and z_I are flagged for I at C (by Lemma 6), $x_I.nxt = y_I$ and $z_I.prv = y_I$ at C.

Statement 4: By induction hypothesis 3, $y_I.prv = x_I$ at the first forward CAS of I. By Lemma 7, $y_I.info$ is always I after that. So, by Lemma 6, no backward CAS of another Info object changes $y_I.prv$ after the first forward CAS of I. Similarly for $y_I.nxt = z_I$. ◀

Consider Fig. 2 and 3. By Lemma 8.3, just before the first forward CAS of I, the nxt and prv field of x_I, y_I and z_I are as shown in step 1. By Lemma 8.1, this CAS changes $x_I.nxt$ as shown in step 2 and the first backward CAS of I changes $z_I.prv$ as shown in step 3. The next lemma follows easily.

▶ **Lemma 9.** *A Node $v \neq head$ that was reachable before is reachable now iff $v.prv.nxt = v$.*

5.3 Part 3: Linearizability

Part 3 of our proof shows that operations are linearizable. The following four lemmas show that there is a linearization point for each move operation. In the following four proofs, $\langle y, \text{-}, z, x, \text{-}, \text{-}\rangle$ denotes the result updateCursor(c) returns on line 44 or 57. Let C_{77} be the configuration before the last execution of line 77 inside that call to updateCursor.

▶ **Lemma 10.** *If moveRight(c) changes c.node from y to z at line 60, there is a configuration during the move when $y.nxt = z$ and y is reachable.*

Proof Sketch. Invariant 1 and Lemma 9 imply that $y \neq head$ is reachable in C_{77}. (Some reasoning is required to see this, since line 77 does two reads of shared memory.) If y is reachable when $y.nxt = z$ on line 85, the claim holds. Otherwise, between C_{77} and line 85, a forward CAS of some Info object I with $I.nodes[1] = y$ made y unreachable. By Lemma 8.4, $y.nxt$ is always $I.nodes[2]$ after the CAS. Since $y.nxt = z$ on line 85, $z = I.nodes[2]$ and, by Lemma 8.3, the claim holds just before the CAS. ◀

▶ **Lemma 11.** *If moveRight(c) returns false, there is a configuration during the move when c.node.val = EOL and c.node is reachable.*

Proof Sketch. Invariant 1 and Lemma 9 imply, in C_{77}, y is reachable and the claim is true. ◀

▶ **Lemma 12.** *If moveLeft(c) changes c.node from y to v at line 49, 53 or 54, there is a configuration during the move when $v.nxt = y$ and v is reachable.*

Proof Sketch. Consider line 53. Since the move does not return on line 46, $x \neq head$. Lemma 9 implies x is unreachable after line 47. By Lemma 8.1, x became unreachable by the first forward CAS of some Info object I with $I.nodes[1] = x$. Since $x.state =$ marked on line 48, a delete created I. By Lemma 8.4, $x.nxt$ is always $I.nodes[2]$ after the forward CAS. Since $x.nxt = y$ on line 47, $y = I.nodes[2]$. Since the read of $y.prv$ returns x

on line 85, the first backward CAS of I did not occur before that read (step 2 of Fig. 2). So, at some configuration C during the move (step 2 of Fig. 2), $I.nodes[0].nxt = I.nodes[2] = y$ and $I.nodes[0]$ is reachable. Since $x.prv$ is always $I.nodes[0]$ after the forward CAS of I, the move sets w to $I.nodes[0]$ on line 51 and then sets $c.node$ to w. So, the claim holds at C. The proofs for line 49 and 54 are similar to the case above and the proof of Lemma 10, respectively. ◀

▶ **Lemma 13.** *If moveLeft(c) returns false, c.node is head.nxt in a configuration during the move.*

Proof Sketch. If moveLeft returns on line 46, the proof is similar to Lemma 10, since $x = head$ and $c.node = head.nxt$ at a configuration during the move. For line 52, the proof is similar to Lemma 12, since $w = head$ and $c.node = head.nxt$ at a configuration during the move. ◀

We now define linearization points. A move is linearized at the step after the configuration defined by Lemma 10, 11, 12 or 13. If there is a forward CAS of an Info object created by an update, the update is linearized at the first such CAS. InitializeCursor and resetCursor are linearized when they read *head.nxt*. Each get, each delete that returns false and each operation that returns invalidCursor is linearized at the first step of the last execution of line 77 inside its last call to updateCursor. Let $(\mathbb{L}, \mathbb{S})$ be an auxiliary variable of type list. When an operation is linearized, the same operation is atomically applied to $(\mathbb{L}, \mathbb{S})$ according to the sequential specification. To prove our linearization is correct, we show in Lemma 14 how the auxiliary variable $(\mathbb{L}, \mathbb{S})$ is accurately reflected in the state of the actual list, implying each operation returns the same response as the corresponding operation on $(\mathbb{L}, \mathbb{S})$.

In Lemma 14, we use abstract values to construct a one-to-one correspondence between Nodes in the list and items in $\mathbb{L}$. The *absVal* of the EOL item is 1. If an item $\mathbb{q}$ is inserted between items $\mathbb{p}$ and $\mathbb{r}$ in $\mathbb{L}$, $\mathbb{q}.absVal = (\mathbb{p}.absVal + \mathbb{r}.absVal)/2$. If $\mathbb{q}$ is inserted before the first item $\mathbb{r}$, $\mathbb{q}.absVal = \mathbb{r}.absVal/2$. The cursor in $\mathbb{S}$ corresponding to Cursor c is denoted $\mathbb{c}$. After the linearization point of a successful operation *op* called with c, *op* might update *c.node*, but $\mathbb{c}$ is updated at the linearization point. To keep track of the value of $\mathbb{c}$, we define a prophecy variable *c.updatedNode*. If a configuration C is after the linearization point of a successful operation *op* called with c but before *op* sets *c.node*, then *c.updatedNode* in C is the Node that *op* would set *c.node* to later. Otherwise, *c.updatedNode* = *c.node*. Since c is a local variable, *c.node* might become out of date when other processes update $\mathbb{c}$. The true location of a cursor c whose *c.upatedNode* is x is

$$realNode(x) = \begin{cases} realNode(x.copy) & \text{if } x.state = \text{copied and } x \text{ is unreachable,} \\ realNode(x.nxt) & \text{if } x.state = \text{marked and } x \text{ is unreachable,} \\ x & \text{otherwise.} \end{cases}$$

We say $realNodePath(x)$ in configuration C is the sequence of Nodes used to define $realNode(x)$ starting with x and ending at a reachable Node. An update is *successful* if it is linearized at a forward CAS.

▶ **Lemma 14.**
1. *If operation op is linearized at step S and terminates with result r, the corresponding abstract operation applied to $(\mathbb{L}, \mathbb{S})$ atomically at S also returns r.*
2. *The sequence of abstract values and values of Nodes that are reachable (excluding head and tail) and of items in $\mathbb{L}$ are equal.*
3. *For each $\mathbb{c}$ in $\mathbb{S}$, $\mathbb{c}.item.absVal = realNode(c.updatedNode).absVal$.*

4. $\mathbb{c}.invIns$ *is true in configuration* C *iff (a) a Node* x *is on* $realNodePath(c.updatedNode)$ *in* C *such that* x *is copied and unreachable, or (b)* C *is between the invocation of an operation op called with c and op's linearization point and op's local variable* $invIns$ *is true.*
5. $\mathbb{c}.invDel$ *is true in configuration* C *iff (a) a Node* x *is on* $realNodePath(c.updatedNode)$ *in* C *such that* x *is marked and unreachable, or (b)* C *is between the invocation of an operation op called with c and op's linearization point and op's local variable* $invDel$ *is true.*

Proof Sketch. Suppose the lemma is true up to step S and C is the configuration before S. We show the lemma is true at the configuration C' after S.

Statement 1: Suppose S is the linearization point of *op* called with c.

Case 1: *op* returns invalidCursor. S is the first step of the last execution of line 77 inside *op*'s last call to updateCursor. In C, *op*'s $invDel$ or $invIns$ is true and $\mathbb{c}.invDel$ or $\mathbb{c}.invIns$ is true by induction hypothesis 4 and 5. So, the abstract operation also returns invalidCursor.

Case 2: *op* is a delete that returns false. S is the first step of the last execution of line 77 inside *op*'s last call to updateCursor. By similar argument to Case 1, $\mathbb{c}.invDel$ is false in C. We show $\mathbb{c}.item.value = \text{EOL}$ in C. In C, $c.updatedNode = c.node$. Invariant 1 and Lemma 9 imply that $c.node$ is reachable in C. In C, $\mathbb{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(c.node).absVal = c.node.absVal$ (by induction hypothesis 3). By induction hypothesis 2, $\mathbb{c}.item.value = c.node.value = \text{EOL}$ in C. So, $\mathbb{c}.item.value = \text{EOL}$ in C and the abstract deletion must also return false.

Case 3: *op* is a move, get or successful update. This case is handled similarly to Case 1 and 2.

Statement 2: By Statement 1, unsuccessful updates change neither $\mathbb{L}$ nor the reachable Nodes. By Lemma 8.1, $\mathbb{L}$ and the reachable Nodes are changed only by the first forward CAS of an Info object. Suppose S is the first forward CAS of an Info object I created by a delete(c). (A similar argument applies to insertBefore.) Since $c.updatedNode = c.node = y_I$ is reachable at C (by Lemma 8.3), $\mathbb{c}.item.absVal = y_I.absVal$ at C (by induction hypothesis 3). Only y_I becomes unreachable at C'. Likewise, only $\mathbb{c}.item$ is removed from $\mathbb{L}$ at C'.

Statement 3: Only linearization points of operations can change $realNode(c.updatedNode)$ or $\mathbb{c}.item$. Suppose S is the linearization point of *op*. We consider different cases.

Case 1: *op* is an initializeCursor or resetCursor that terminates. Then, step S is a read of $head.nxt$ on line 63 or 68. By Statement 1, S sets $\mathbb{c}.item$ to the first item in $\mathbb{L}$. Let x be the value of $head.nxt$ in C'. The $absVal$ of the first item in $\mathbb{L}$ is $x.absVal$ in C' (by Statement 2) and $\mathbb{c}.item.absVal = x.absVal$ in C'. Since *op* sets $c.node$ to x on line 63 or 68, $c.updatedNode = x$ in C'. In C', $\mathbb{c}.item.absVal = x.absVal = realNode(x).absVal = realNode(c.updatedNode).absVal$.

Case 2: *op* is called with c and returns invalidCursor or *op* is a delete(c) that returns false or *op* is a get(c) that terminates. Then, S is the first step of the last execution of line 77 inside *op*'s last call to UPDATECURSOR. By Statement 1, S does not change $\mathbb{c}.item$. By induction hypothesis 3, $\mathbb{c}.item.absVal = realNode(c.updatedNode).absVal$ in C. Since S does not change $realNode(c.updatedNode)$, the same equality holds in C'.

Case 3: *op* is a move or successful update. This case is handled similarly to Case 1.

Statement 4 and 5: We show Statement 4 is true. The proof of Statement 5 is symmetric. It is easy to show that the only steps S that we must consider are linearization points of operations, executions of line 79, 80 or 83 and invocations of an operation called with c. Suppose S is the linearization point of an operation *op* called with c. In C', $\mathbb{c}.invIns$ and Statement 4.b are false. It is easy to show that, in C', $c.updatedNode$ is reachable and it is the only Node on $realNodePath(c.updatedNode)$. So, Statement 4.a is false. The rest of cases are handled similarly. ◀

Linearizability of our implementation follows from Lemma 14.1.

6 Amortized Analysis

Our amortized analysis gives an upper bound on the total number of steps performed by all operations in any finite execution. A Cursor is *active* if it has been initialized, but not yet destroyed. Let $\dot{c}(op)$ be the maximum number of active Cursors at any configuration during operation op. We prove that the amortized complexity of each update op is $O(\dot{c}(op))$ and each move is $O(1)$. More precisely, for any finite execution α, the total number of steps in α is $O(\sum_{op \text{ is an update in } \alpha} \dot{c}(op) + \sum_{op \text{ is a move in } \alpha} 1)$. It follows that the implementation is non-blocking. The analysis (about 20 pages long in [19]) is quite complex, so we only provide the intuition here. Parts of it are similar to the analysis of search trees in [6] but the parts dealing with Cursors and moves are original. We simplified the analysis using the potential method and show how to generalize the analysis of [6] to handle operations that flag more than two nodes.

We first bound the number of iterations of line 77–83. Each update op deletes or replaces at most one Node. Any Cursor c whose true location (as defined in Sec. 5) is at that Node when op is performed will have to perform one iteration of line 77–83 when updateCursor(c) is next called to follow the *nxt* or *copy* pointer of the Node. Since there are at most $\dot{c}(op)$ such cursors, the total number of iterations of line 77–83 in the execution is at most $\sum_{op \text{ is an update}} \dot{c}(op)$.

Each iteration of line 18–30 or 32–42 inside an update is called an *attempt*, which is *successful* if it returns on line 20, 30, 34 or 42, or *unsuccessful* otherwise. Excluding calls to updateCursor (which have already been accounted for), each attempt of an update takes $O(1)$ steps and, each move and each get operation take a total of $O(1)$ steps. It follows that the amortized complexity of a move and a get is $O(1)$. It remains to prove that the total number of unsuccessful attempts in the execution is $O(\sum_{op \text{ is an update}} \dot{c}(op))$. An attempt is unsuccessful because one of the Nodes to be flagged is either (1) observed to be marked or copied when checkInfo returns false on line 92 or (2) flagged by another update.

First, consider attempts that fail for reason (1). Consider a Cursor c that is active when an update op sets $x.state$ to copied or marked. This causes at most two attempts of c's updates to fail: if an attempt of an update op' fails when reading $x.state$ at line 92, line 89 of the next attempt ensures op is completed and no subsequent attempt reaches x. To pay for these attempts, op stores $2\dot{c}(op)$ of potential when op sets $x.state$. There is one other possibility: an operation op' might be called with a Cursor that is created *after* op set $x.state$ to marked or copied. Again, at most two attempts of op' might fail because of reading $x.state$ at line 92. To pay for these attempts, op' stores $2\dot{c}(op')$ of potential when it is invoked, since there are at most $\dot{c}(op')$ reachable Nodes that are marked or copied when op' begins. Thus, the total number of attempts that fail for reason (1) is $O(\sum_{op \text{ is an update}} \dot{c}(op))$.

Bounding the number of attempts that fail for reason (2) is the most intricate part of the analysis. An attempt *att* of an update may fail because a Node it wishes to flag gets flagged by an attempt *att*′ of another operation, causing *att*'s test at line 88 or 94 to fail or *att*'s flag CAS on line 99 to fail. If *att*′ were guaranteed to succeed in this case, the analysis would be simple. However, *att*′ itself may also fail because it is blocked by the attempt of a third operation, and so on. Since a successful flag CAS might belong to an unsuccessful attempt, such a step does not store any potential. However, $O(\dot{c}(op))$ of potential is stored

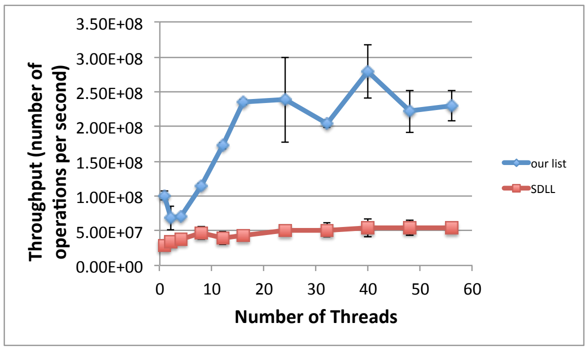

Figure 6 Comparison of our list and the list in [21].

(a) when an update *op* is invoked, (b) when a forward or backward CAS step of *op* succeeds and (c) when the *status* of an Info object created by *op* is set to committed. We prove that this potential is sufficient to pay for any attempt that fails for reason (2). Thus, the total number of attempts that fail for reason (2) is also $O(\sum_{op \text{ is an update}} \dot{c}(op))$. It follows the amortized complexity of an update *op* is $O(\dot{c}(op))$.

7 Concluding Remarks

A correctness proof is essential for data structures such as ours since it is not possible to test all possible executions. Writing detailed correctness proofs helped us to correct bugs in earlier versions and then verify the correctness of our list. It also helped us to simplify the pseudo-code and improve its complexity.

Our amortized bound of $O(\dot{c}(op))$ for an update *op* is quite pessimistic: the worst case happens only if concurrent updates are scheduled in a very particular way. We expect our list would have even better performance in practice. Our experimental results suggest that on an Intel Xeon multi-core machine, a Java implementation of our list scales well and also outperforms the list in [21] (SDLL). We used our own Java implementations for both our list and SDLL. Each data point in Figure 6 is the average of eight 4-second trials in which each thread continuously performs on-average 100 move operations and then one update operation on a list with 200 items. Our results show that the throughput of SDLL is not improved when the number of threads is increased, which agrees with the empirical results presented in [21].

In our approach, as in [3, 4, 5, 8, 18], updates create Info objects and duplicate Nodes, which induces some overhead. Despite such overheads, empirical evaluations in [4, 5, 18] and here confirm the practicality and scalability of this technique.

Though moves have constant amortized time, they are not wait-free. For example, if cursors c and c' point to the same node, a moveLeft(c) may never terminate if an infinite sequence of insertions at c' succeed, since the updateCursor of the move could run forever.

Future work includes designing shared cursors. Generalizing our coordination scheme could provide a simpler way to design non-blocking data structures. Although the proof of correctness and analysis is complex, it is modular, so it could be applied more generally. Our help routine gives a general way of coordinating operations that make several changes to a data structure. Parts 1 and 2 of the proof are primarily about this routine and could be reused for other data structures. Detailed arguments about linearizability of the operations (Part 3 of the proof) would likely depend more on the data structure being implemented.

Acknowledgements. I thank my supervisor, Eric Ruppert, for his great guidance and advice, and the management, staff, and facilities of the Intel Manycore Testing Lab.[1]

References

1 Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.

2 Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, SPAA'93, pages 261–270, 1993.

3 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, PODC'13, pages 13–22, 2013.

4 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming*, PPOPP'14, pages 329–342, 2014.

5 Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proceedings International Conference on Principles of Distributed Systems*, OPODIS'11, pages 207–221, 2011.

6 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, PODC'14, pages 332–340, 2014.

7 Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, PODC'12, pages 115–124, 2012.

8 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC'10, pages 131–140, 2010.

9 Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, July 1990.

10 Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, PODC'02, pages 260–269, 2002.

11 Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition.* 2010.

12 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC'02, pages 265–279, 2002.

13 Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000. See `http://babelfish.arc.nasa.gov/trac/jpf`.

14 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[1] `http://www.intel.com/software/manycoretestinglab`
`http://www.intel.com/software`.

15 Jikuan Hu and Weiqing Wang. Algorithm research for vector-linked list sparse matrix multiplication. In *Proceedings of the 2010 Asia-Pacific Conference on Wearable Computing Systems*, APWCS'10, pages 118–121, 2010.

16 Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, SPAA'03, pages 314–323, 2003.

17 Matthias Pfeffer, Theo Ungerer, Stephan Fuhrmann, Jochen Kreuzinger, and Uwe Brinkschulte. Real-time garbage collection for a multithreaded Java microcontroller. *Real-Time Systems*, 26(1):89–106, January 2004.

18 Niloufar Shafiei. Non-blocking Patricia tries with replace operations. In *Proceedings of the 33rd International Conference on Distributed Computing Systems*, ICDCS'13, pages 216–225, 2013.

19 Niloufar Shafiei. *Non-blocking data structures handling multiple changes atomically.* PhD thesis, Department of Electrical Engineering and Computer Science, York University, Toronto, Canada, August 2015.

20 Håkan Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming*, 39(6):694–716, 2011.

21 Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

22 John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, PODS'92, pages 212–222, 1992.

23 John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 214–222, 1995.