Theory Comput Syst (2013) 52:729–762
DOI 10.1007/s00224-012-9420-5

Built-in Coloring for Highly-Concurrent Doubly-Linked Lists

Hagit Attiya · Eshcar Hillel

Published online: 1 August 2012

Abstract This paper presents a novel approach for highly-concurrent nonblocking implementations of doubly-linked lists, based on dynamically maintaining a *coloring* of the nodes in the list. In these implementations, operations on non-adjacent nodes in the linked-list proceed without interfering with each other. Roughly speaking, the operations are implemented by acquiring nodes in the operation's data set, in the order of their colors, and then making the changes atomically. The length of waiting chains is restricted, thereby increasing concurrency, because the colors are taken from a small set. Operations carefully update the colors of the nodes they modify, so neighboring nodes in the list have different colors. A helping mechanism ensures progress in small neighborhoods of processes that keep taking steps.

We use this approach in two new algorithms: CAS-Chromo uses an unary conditional primitive, CAS, and allows insertions anywhere in the linked list and removals only at the ends, while DCAS-Chromo allows insertions and removals anywhere but uses a stronger primitive, DCAS.

Keywords Asynchronous shared memory · Concurrent data structures · Local nonblocking implementations · Doubly-linked list · Double-ended queue · Priority queue

A preliminary version of this paper has appeared in the proceedings of the 20th International Symposium on Distributed Computing (DISC'06), pp. 31–45.

H. Attiya (✉) · E. Hillel
Department of Computer Science, Technion, Haifa, Israel
e-mail: hagit@cs.technion.ac.il

E. Hillel
Yahoo! Labs, Matam Advanced Technology Park, Haifa, Israel
e-mail: eshcar@yahoo-inc.com

1 Introduction

Many core problems in asynchronous multiprocessing systems can be captured as *concurrent data structures*—abstract data structures that are concurrently accessed by asynchronous processes. A prominent example is provided by list-based data structures: A *double-ended queue* (*deque*) supports operations that insert and remove nodes at the two ends of the queue; it provides a producer-consumer job queue [3]. A *priority queue* can be implemented as a doubly-linked list with removals only at the ends, while nodes can be inserted anywhere at the queue; it can be used to queue process identifiers for scheduling purposes. Finally, a generic *doubly-linked list* (hereafter, called simply a *linked list*) allows insertions and removals of nodes anywhere in the list.

Concurrent data structures are implemented by applying *primitives*, provided by the hardware or the operating system, to memory locations. These primitives usually include CAS (*compare&swap*) or its multi-location variant, *k*CAS. These implementations are hard to get right, and even for relatively simple data structures, like deques, significant compromises are made: In some implementations, removed nodes remain in the list [14], others statically limit the data structure's size [17] or do not allow concurrent operations on both ends of the queue [21]. Even when DCAS (i.e., 2CAS) is used, existing implementations either are inherently sequential [11, 12] or allow access to chains of removed nodes [9].

Implementing concurrent data structures is simpler if an arbitrary number of locations can be accessed atomically. For example, removing a node from a doubly-linked list is easy if one can atomically access three nodes—the node to be removed and the two nodes before and after it (cf. [9]).

Known methods, such as [7, 24, 27], simulate this atomicity in software by using CAS to acquire the nodes, one by one, and *help* processes that previously acquired a desired node until it is released. In this way, some operation always completes within a finite number of steps. In these methods, nodes are acquired according to the order of their memory addresses. Unfortunately, as Sect. 6 shows, these implementations have long waiting chains in some symmetric scenarios, creating interference among operations and reducing the throughput. Although waiting chains can be shortened by breaking symmetry, using colors [1, 4], randomization [13, 23] or DCAS [5], their length is still non-constant (see Sect. 6).

This paper presents a novel approach for reducing the length of waiting chains in concurrent implementations of linked lists. Our approach exploits the fact that operations on the linked list access it in a predictable, well-organized manner, namely, a small number of consecutive nodes in the list. Operations acquire nodes not by the order of their memory addresses, but by the order of built-in *colors*, taken from a small set, which are associated with the nodes. To avoid deadlocks and ensure short waiting chains, the operations preserve the legality of the nodes' colors when they modify the linked list; this is possible since the implementation initializes the data structure and provides operations that are the only means for manipulating it.

Our implementations are *local nonblocking*, namely, they ensure that when operations access distant parts of the data structure, or are separated in time, they do not interfere with each other. Formalizing this notion relies on defining the *distance*

between operations to be the length of the shortest path between them in the *conflict graph* (defined in Sect. 2.3). In a *d-local nonblocking* implementation, whenever an operation *op* takes an infinite number of steps, some operation, within distance d from *op*, completes. This guarantees progress in components of the conflict graph with diameter d, and ensures that operations are effectively *isolated* from other operations at distance $>d$.

The high-level idea is simple: a 3-coloring of the nodes determines the order in which an operation acquires the nodes. After acquiring the nodes needed for a list operation, its changes are applied in isolation. Equally-colored nodes are acquired by their list order (from left to right) to ensure that there are no deadlocks and that some operation makes progress at any time.

Our first algorithm, CAS-Chromo, allows removals only at the ends of the linked list and uses CAS; it can be used as a simple deque or a priority queue. Since the list is 3-colored, we can show that CAS-Chromo is 7-*local nonblocking*, namely, an operation is delayed only due to operations on nodes close to its own nodes on the linked list. When insertions are limited to occur at the ends (i.e., a deque), the analysis can be further refined to show that the algorithm is 3-local nonblocking; in particular, operations at the two ends of a deque containing at least three nodes do not delay each other.

Our second algorithm, DCAS-Chromo, allows insertions and removals anywhere in the list. Removals from the middle of the linked list are more difficult: a remove operation must acquire three consecutive nodes, two of which may have the same color. Thus, removing a node might entail recoloring one of its neighbors while ensuring its neighbor's color is not changed concurrently. To do this without creating hold-and-wait chains we employ DCAS to atomically acquire two nodes with the same color. DCAS-Chromo is 5-*local nonblocking*, namely, only operations on nodes that are separated by fewer than four nodes may delay each other.

The rest of this paper is organized as follows. Section 2 presents the model of an asynchronous shared-memory system, specifies the doubly-linked list data structure, and defines locality properties in a dynamic setting. CAS-Chromo, a CAS-based priority queue, is presented in Sect. 3.1. Section 3.2 outlines the modifications needed to obtain DCAS-Chromo, a DCAS-based linked list supporting removals anywhere. The correctness proof of both algorithms is given in Sect. 4, while their locality properties are analyzed in Sect. 5. Finally, we discuss related work in Sect. 6 and conclude, in Sect. 7.

2 Preliminaries

2.1 A Model for Shared-Memory Systems

We consider a standard model for a shared memory system [6] in which a finite set of *asynchronous processes* $p_1, \ldots, p_n$ communicate by applying *primitive* operations to shared *memory locations*, $l_1, \ldots, l_m$. A *configuration* specifies the local state of each process and the value of each memory location. In the (unique) *initial configuration*, every process is in its initial state and every location contains its initial value.

```
boolean CAS(l, exp, new) {
    // Atomically
    if l = exp then
        l ← new
        return TRUE
    return FALSE
}
```

```
boolean DCAS(l1, l2, e1, e2, n1, n2) {
    // Atomically
    if l1 = e1 and l2 = e2 then
        l1 ← n1
        l2 ← n2
        return TRUE
    return FALSE
}
```

Fig. 1 The CAS and DCAS primitives

An *event* is a computation step by a single process, consisting of some local computation and the application of a primitive to the memory. We employ the following primitives: READ(l_j) returns the value v_j in location l_j; WRITE(l_j, v) sets the value of location l_j to v; CAS(l_j, exp, new) writes the value *new* to location l_j if its value is equal to *exp*, and returns a success or failure indication; DCAS is similar to CAS, but operates on two independent memory locations (see Fig. 1).

An *execution interval* α is a (finite or infinite) alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \ldots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \ldots$. An *execution* is an execution interval in which C_0 is the unique initial configuration.

A *data structure type* supports a set of operations that provide the only means to manipulate it. The sequential specification of an *operation* indicates how the data structure is modified when operations are applied in a serial manner (in isolation). It is given as a sequence of read and write instructions executed on a set of items, which constitute the operation's *data set*.

An *implementation* of a data structure type provides a specific representation for the data of an instance as a set of memory locations, and protocols that processes follow to carry out its operations, defined in terms of primitives applied to memory locations. In order to apply an operation, a process executes the protocol associated with the operation.

The *interval of an operation op* is the execution interval that starts at the first event of *op* and ends at the last event of *op*, if there is one. If the operation does not complete, its interval is infinite, and we say that it is *pending*.

Two operations *overlap* if their intervals overlap.

An execution is *serial* if operations do not overlap; this means that every operation is executed to completion before another operation starts.

Two executions are *equivalent* if every process in these executions issues the same operations in the same order and gets the same result for each operation.

We require the implementation to be *linearizable* [16], that is, any execution can be extended by discarding some pending operations and completing the others, such that the extended execution is equivalent to some serial execution, called its *linearization*, which preserves the order of non-overlapping operations. Note that an execution could have several linearizations.

```
PushLeft(nd) {
    nd.right ← LeftAnchor.right
    nd.left ← LeftAnchor
    LeftAnchor.right.left ← nd
    LeftAnchor.right ← nd
    return SUCCESS
}
```

(a) PushLeft: Insert *nd* to the right of the left anchor and return SUCCESS

```
PopLeft() {
    if empty list then
        return EMPTY
    subject ← LeftAnchor.right
    LeftAnchor.right ← subject.right
    LeftAnchor.right.left ← LeftAnchor
    subject.right ← ⊥
    subject.left ← ⊥
    return SUCCESS
}
```

(b) PopLeft: If the linked list is not empty, then remove the *subject* next to the left anchor and return SUCCESS; otherwise, return EMPTY and the linked list is unchanged

```
InsertRight(source, nd) {
    if source is not in the list or
        source is right anchor then
        return INVALID
    nd.right ← source.right
    nd.left ← source
    source.right.left ← nd
    source.right ← nd
    return SUCCESS
}
```

(c) InsertRight: If *source* is a node in the list different from the right anchor, then insert *nd* to the right of *source* and return SUCCESS; otherwise, return INVALID and the linked list is unchanged

```
Remove(source) {
    if source is not in the list or source is an
        anchor then
        return INVALID
    source.left.right ← source.right
    source.right.left ← source.left
    source.right ← ⊥
    source.left ← ⊥
    return SUCCESS
}
```

(d) Remove: If *source* is a node in the list different from an anchor, then remove *source* from the linked list and return SUCCESS; otherwise, return INVALID and the linked list is unchanged

Fig. 2 Sequential specification of the linked list operations

2.2 Doubly-Linked Lists

This paper considers a *doubly-linked list* data structure; the items are the *nodes* composing the linked list. Each node has links to its left and right neighboring nodes. Two special *anchor* nodes serve as the leftmost and rightmost nodes in the doubly-linked list, denoted *LA* and *RA*; they cannot be removed from it, and have no left link or no right link, respectively. A node is *valid* if it is either an anchor, or both its left link and right link pointers are not null.

Figure 2 provides the sequential specification of the operations that can be applied to the data structure, following the description of deque operations [2].

We concentrate on the following operations: *PushLeft*, *PopLeft* operations, which inserts or removes, respectively, the node next to (to the right of) the left anchor. The *InsertRight* operation, applied to a *source* node in the linked list, inserts a node to the right of the source node. Finally, a *Remove* operation, applied to a source node in the linked list, removes the source node. *PushRight*, *PopRight* and *InsertLeft* operations are defined analogously.

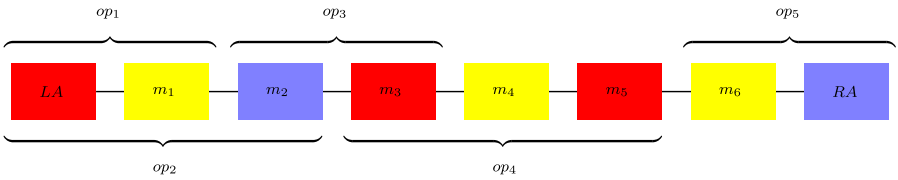


Fig. 3 Overlapping operations of a doubly-linked list, *LA* (*RA*) is the left (right) anchor

2.3 Locality Properties

The locality of an implementation is defined relative to a *conflict graph*, capturing the distance between overlapping operations. In this graph, the nodes are concurrent operations and there is an edge between two operations with intersecting data sets. Formally defining the data set of an operation requires care since the data structure—and hence the data sets of its operations—are dynamic. This means that the state of the linked-list in a configuration is not uniquely determined when there are several pending operations.

Consider, for example, Fig. 3, in which op_1 is a PushLeft operation, op_2 is a PopLeft operation, op_5 is a PushRight operation, op_3 inserts a new node to the right of m_2, and op_4 removes m_4. Let $\alpha_1\alpha_2\alpha_3$ be the following execution of these operations: in the execution interval α_1 every operation takes steps, but makes no changes to the memory; then, in the execution interval α_2 op_2 takes steps alone until it successfully removes m_1; finally, in the execution interval α_3 only op_1 takes steps until it completes. Let C_1, C_2, and C_3 be the configurations at the end of the execution intervals α_1, α_2, and α_3, respectively.

Observe that the data set of op_1 in C_1 is $\{LA, m_1\}$, but after m_1 is removed, in C_2, the data set of op_1 is $\{LA, m_2\}$; altogether, the data set of op_1 is $\{LA, m_1, m_2\}$. The next definition captures such scenarios.

Definition 1 Let C be the configuration after an execution prefix α, in which an operation op is pending. The *data set of op in C* is the union, over all possible linearizations α_S of α that do not include op, of the set of items accessed by op when executed after α_S. The *data set of op* is the union of its data sets in all the configurations during its execution interval.

The *conflict graph* of a configuration C is an undirected graph in which vertices represent operations, and an edge connects two operations whose data sets intersect. The conflict graph of an execution interval α is the union of the conflict graphs of all configurations C in α; that is, the vertices (edges, respectively) in the graph are the union of the vertices (edges, respectively) of all these conflict graphs.

The *distance* between two operations, $op \neq op'$, in a conflict graph, is the length (in edges) of the shortest path between op and op'; the distance from an operation to itself is zero. The distance between two operations is one if their data sets intersect; if there is no path between the operations, the distance is infinity. The *d-neighborhood of an operation op* contains all the operations within distance d from op.

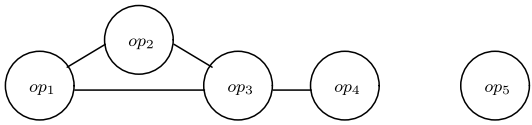


Fig. 4 The conflict graph of the execution interval $\alpha_1\alpha_2\alpha_3$ of the operations from Fig. 3

Figure 4 depicts the conflict graph of $\alpha_1\alpha_2\alpha_3$, the interval of the operation op_1 from Fig. 3. In configuration C_1, the distance between op_1 and op_2 is one, the distance between op_1 and op_3 is two, the distance between op_1 and op_4 is three, and the distance between op_1 and op_5 is ∞. After op_2 completes in configuration C_2, the distance between op_1 and other operations is decreased by one, since at this configuration the data set of op_1 includes m_2. During the interval of op_1, the distance between op_1 and op_3 is one, the distance between op_1 and op_4 is two, and the distance between op_1 and op_5 remains ∞. Therefore, the 1-neighborhood of op_1 includes op_1, op_2, and op_3; the 2-neighborhood of op_1 includes op_1, op_2, op_3 and op_4; there is no d such that op_5 is in the d-neighborhood of op_1.

The next definition ensures progress in the neighborhood of an operation.

Definition 2 An algorithm is *d-local nonblocking* if whenever a process takes an infinite number of steps in an operation op, then some operation in the d-neighborhood of op completes.

3 Algorithms Using Built-in Coloring

3.1 CAS-Chromo: Priority Queue and Deque

CAS-Chromo is a doubly-linked list implementation that allows insertions anywhere (*PushLeft*, *PushRight*, *InsertRight*, and *InsertLeft*) and removals only at the ends (*PopLeft*, and *PopRight*); see Fig. 2. Our description focuses on the *PushLeft* and *PopLeft* operations.

3.1.1 Overview

We follow a known scheme [7, 24, 27] for systematically deriving implementations of concurrent data structures: An operation first *acquires* the nodes in its data set (ACQUIRE phase), and then applies its changes atomically on these nodes (APPLY phase); finally, the operation releases the nodes (RELEASE phase). As in a lock-based solution, other operations cannot modify a node when it is owned by an operation. During the ACQUIRE phase, an operation may own one or more nodes while waiting for another operation to release a node. The latter operation might also be waiting for a third operation to release a node, leading to a *hold-and-wait chain* of operations.

The key novelty of our approach is in acquiring nodes *by colors*, in order to shorten the hold-and-wait chains. The nodes of the linked-list are legally colored (so that neighboring nodes have different colors) with three ordered colors, $c_1 < c_2 < c_3$. (In

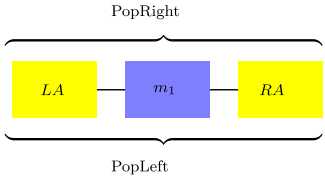


Fig. 5 Example: two operations competing on a single node; anchors have the same color

Fig. 3, the items are colored with three colors, yellow < blue < red.) Since an operation acquires its data set in an increasing order of colors, and the colors of neighboring nodes are different, hold-and-wait chains have (strictly) *increasing* colors from a small set, and hence, are short.

Push and insert operations access exactly two neighboring nodes in the list, which must have different colors. Pop operations access three consecutive nodes, two of which may have the same color. To avoid hold-and-wait cycles, pop operations acquire monochromatic nodes according to their order in the list, from left to right.

For example, consider a doubly-linked list containing a single node as depicted in Fig. 5, where the two anchors have the same color. When the *PopLeft* and *PopRight* operations compete on removing the single node in the list, both of them try to acquire the left anchor first, and at least one of them (that succeeds in acquiring the left anchor) succeeds in popping the last item. Moreover, pop operations occur only at the ends of the list, and hence, each of them adds at most one edge to a hold-and-wait chain (one edge at each end of the chain) in addition to at most three edges connecting operations that have conflicts on nodes with increasing colors. This is used to show that the length of a hold-and-wait chain is at most 5 (see Theorem 4).

Helping The simple algorithm described so far may block if a process stops taking steps while owning a node. An operation op_1 is *blocked* if one of its nodes is already owned by another *blocking* operation op_2. *Helping* guarantees that some operation makes progress at any time while preserving the locality properties of the implementation. This means that instead of waiting, the process executing op_1 helps op_2 to complete by executing its steps. Helping is recursive: if, while helping op_2, the process executing op_1 discovers that op_2 is blocked by a third operation op_3, then the process helps op_3 to complete, and so on. In this way, *hold-and-help* chains replace hold-and-wait chains, and it is possible to prove that their length is bounded by the number of colors (see Lemma 4). For example, assume that operation op_3 in Fig. 3 owns m_2 and then tries to acquire m_3, with color red. If m_3 is already held by op_4, then op_3 helps op_4. However, red is the largest color, implying that op_4 already owns all the nodes in its data set. Thus, when helping op_4, op_3 only needs to apply the changes of op_4, and not recursively help additional operations.

Synchronization We use two known techniques to guarantee that processes work in harmony, and do not override the modifications done by other processes. An operation acquires the nodes it needs to modify, and it uses CAS for all modifications it applies to these nodes, to verify that the nodes have not changed since they were acquired. Acquiring nodes is required to avoid conflicts between different operations; changing the nodes by applying CAS is required to avoid races between different processes executing the same operation, due to the helping mechanism.

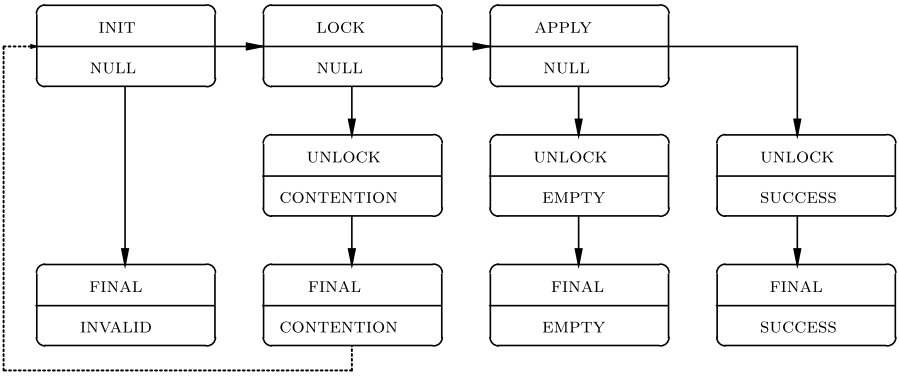


Fig. 6 The state transitions of an operation: the lower part of the state is the *result* of the current round; the dashed line indicates a new round. The best-case scenario, encountering no contention, appear at the top. If the list is empty during a pop operation, then the operation completes without changing the linked list. If the source node is removed during the INIT phase of an operation, then the operation need not apply its changes. If there is contention during the ACQUIRE phase, the operation releases its nodes and it is re-invoked

Maintaining a Legal Coloring Since operations apply their changes in exclusion, they can ensure that the coloring is legal at all times, by careful color changes during the APPLY phase.

The operations use a temporary color $c_0 < c_1$, which is white in the figures, to simplify the task of maintaining the coloring legal: In the *PushLeft* operation, the color of the new node is c_0. After the new node is in the list, and while the data set is not released yet, the new node is assigned with a color different than its neighbors. In the *PopLeft* and *PopRight* operations, after the data set is acquired, the color of the right neighbor of the node to be removed is changed to c_0. After the node is removed, the color of the right neighbor is changed to be different from its neighbors' colors.

Dynamic Aspects The algorithm accommodates a dynamic data structure, that is, after an operation reads the data set and verifies that the nodes are valid, another operation may modify the nodes and even the list structure, changing the data set of *op*. In a manner similar to [15], a *data set memento*, holding a view of the data set when the operation is started, traces inconsistencies in the data set due to changes by concurrent operations. If, while acquiring the data set, an operation discovers that a node in its data set memento is invalid or inconsistent with its memento, it restarts. That is, the operation skips the APPLY phase and goes directly to the RELEASE phase where it releases all the nodes it already acquired and re-invokes the operation. Otherwise, the operation completes its ACQUIRE phase, and the nodes it has acquired are consistent with the data set memento, so the operation can continue with the APPLY phase as in a static scheme.

This causes the operation to go through several *rounds*, trying to acquire the nodes, until the operation *completes*. Figure 6 shows the state transition diagram of one round of an operation.

```
define LEFT = 0, MIDDLE = 1, RIGHT = 2
define Phase = {INIT, ACQUIRE, APPLY, RELEASE, FINAL}
define Result = {NULL, SUCCESS, CONTENTION, INVALID, EMPTY}

structure State {int seq, Phase phase, Result result} // seq—for repeated rounds
structure NodePtr {Node node, int aba}
structure Color {int c, int aba}
structure Owner {Operation op, int seq, int aba} // seq—operation's round number

structure Node {                    structure NodeMemento {
    Data     data,                      Node     node,
    NodePtr left,                       NodePtr left,
    NodePtr right,                      NodePtr right,
    Color    color,                     Color    color
    Owner    owner                  }
}

class Operation {
    State              state // initially ⟨0,INIT,NULL⟩
    Node               source // in push/pop operations the source is an anchor
    Node               subject // either the new node or the node to be removed
    NodeMemento[3] datasetMemento
    int[3]             colorSet
}
```

Fig. 7 Types, structures and classes definitions

3.1.2 Data Structures

Figure 7 lists the main types and data structures used in the algorithm.

Operations are objects, whose structure and behavior are defined in the *Operation* class. An operation object is initialized with all the data required for its execution, specifically the source node on which the operation is applied, and the subject node to be inserted to the list (in insert or push operations).

Nodes are also objects. In addition to its *data* attribute, a node contains two pointers *left* and *right*, a *color* and an *owner*. The owner keeps a reference to the operation instance that owns the node, to facilitate helping. A node memento is composed of a reference to the node itself, *node*, and a copy of the node's meta data except for the owner, so the node is consistent with its memento even if the node is acquired and released arbitrarily, as long as the other attributes do not change.

Due to helping, several processes may execute the same operation op; these are the *executing processes* of op. One of the executing processes—the one that first called the execute method of op—is the *initiator* of op.

Since an operation may have several rounds, its execution goes through alternating phases of acquiring and releasing nodes. The *state* of an operation is a tuple $\langle seq, phase, result\rangle$: *seq* is the *round number*, an integer, initially 0, that is incremented when the initiator re-invokes the operation; *phase* is the scheme phase within the round, set to INIT at the beginning of a round; *result* is the result of the current round, set to NULL at the beginning of a round.

The CAS primitive suffers from the ABA problem [18]; namely, a process p may read a value A from some memory location l, then other processes change l to B and then back to A, later p applies CAS on l and the comparison succeeds whereas it should have failed. We avoid this problem by associating each attribute with a monotonically increasing *ABA counter*. The attribute and the counter fit into a single memory location and are manipulated atomically; the counter is incremented whenever the attribute is updated. Assuming that the counter has enough bits, the CAS succeeds only if the counter has not changed since the process read the attribute.

3.1.3 Implementation

Pseudocodes 1, 2, 3 and 4 present the code for CAS-Chromo. The reserved word *self* in the pseudocode denotes the operation object of the operation whose code is being executed.

The initiator starts the execution with the execute method and as long as it suffers from contention and is unable to complete (line 15), the process repeatedly tries to clear the attributes (line 3) and re-invoke the operation. It generates the new data set memento (line 11), and then "helps" itself to follow the scheme (line 13): acquire nodes in its data set (line 20), apply its changes (line 23), and release the data set (line 25).

In the acquireColor method, an operation op repeatedly tries to acquire all nodes with a given color c in its current data set memento; The operation reads the owners of the nodes (line 36); after verifying the nodes are valid and consistent with their mementos (line 37), op tries to acquire the nodes (line 40). If op discovers that none of these nodes is owned by another operation, when failing to acquire them, it simply retries to acquire their nodes. Otherwise, op finds that a node in its data set is owned by another, blocking operation op' (line 45), and helps op' to complete (line 46).

During the execution of an operation, a process applies validity checks to ensure the consistency of the execution, as well as other properties of the implementation, such as locality. We now describe these tests and their motivation, and explain what steps are taken in case of a failure, i.e., when the result of the test is negative.

Before helping op', the executing process of op verifies (again) that the nodes are consistent with their mementos (line 43). This ensures that the color of the node did not decrease, which is crucial for the locality properties of the algorithm. Lemma 4 shows that the length of helping chains is bounded by the number of colors, by proving that a process helps along a chain with monotonically increasing colors. Without this verification, a process may help along an unbounded chain with colors alternately increasing and decreasing.

The state attribute is used to synchronize the executing processes of an operation. It is possible that a delayed (slow) process, executing a previous round, tries to acquire nodes that are associated with this previous round or releases nodes that were re-acquired in the current round. Therefore, an executing process verifies, before acquiring a node, that the round number of its execution matches the one in the state of the operation (line 38). Furthermore, if an executing process detects inconsistency between the node and the operation's memento (line 37 or line 43) then it violates

Pseudocode 1 CAS-Chromo

```
 1: Result execute() {
 2:     do
 3:         clear memento and color set
 4:         toInitState()
 5:         if source is invalid then
 6:             owner ← source.owner
 7:             owner.op.help(owner.seq)
 8:             toFinalInvalidState()
 9:         if state.result = INVALID then return
10:         else // new round
11:             cloneDataset()
12:             toAcqState()
13:             help(state.seq) // help myself
14:             toFinalState()
15:     while state.result = CONTENTION
16:     return state.result
17: }

18: help(int seq) {
19:     for each color c by increasing order do
20:         acquireColor(c,seq)
21:     toApplyState(seq)
22:     if state.phase = APPLY then
23:         applyChanges()
24:         toReleaseState()
25:     releaseDataset(seq)
26: }

27: releaseDataset(int seq) {
28:     for each Node nd in seq-th memento do
29:         owner ← nd.owner
30:         if owner = ⟨self, seq, t⟩ then and state.phase = RELEASE
31:             CAS(nd.owner, owner, ⟨⊥, ⊥, t + 1⟩)
32: }

33: acquireColor(Color c, int seq) {
34:     {nd_i} ← c-colored nodes in seq-th memento
35:     while true do
36:         {owner_i} ← get all owners from {nd_i}
37:         checkNodes({nd_i},seq)
38:         if state != ⟨seq, ACQUIRE, NULL⟩ then return
39:         for each nd_i from left to right do // acquire all equally colored nodes
40:             if owner_i = ⟨⊥, ⊥, t⟩ then CAS(nd_i.owner, owner_i, ⟨self, seq, t + 1⟩)
            // check if owns all c-colored nodes or blocked
41:         owner ← get owner from {nd_i}// returns self only if the operation owns all nodes
42:         if owner.op = self then return // succeeded
43:         checkNodes({nd_i},seq)
44:         if state != ⟨seq, ACQUIRE, NULL⟩ then return
45:         if owner.op != ⊥ then
46:             owner.op.help(owner.seq) // blocked by owner.op—help it
47: }

48: checkNodes(Set⟨Node⟩ {nd_i}, int seq) {
49:     if invalid node or inconsistent memento then
50:         for each i do // "touch" the ABA counter of the owner
51:             l_i ← nd_i.owner
52:             CAS(nd_i.owner, l_i, ⟨l_i.op, l_i.seq, l_i.aba + 1⟩)
53:         toReleaseContentionState(seq)
54: }
```

Pseudocode 2 CAS-Chromo: Code for the clone methods

```
56:  PushLeft::cloneDataset() {
         // subject is pushed, source is the left anchor
57:      cloneNode(source, LEFT)
58:      cloneNode(subject, MIDDLE)
59:      right ← datasetMemento[LEFT].right.node
60:      cloneNode(right, RIGHT)
61:  }

62:  PopLeft::cloneDataset() {
         // subject is popped, source is the left anchor
63:      cloneNode(source, LEFT)
64:      subject ← datasetMemento[LEFT].right.node
65:      cloneNode(subject, MIDDLE)
66:      right ← datasetMemento[MIDDLE].right.node
67:      cloneNode(right, RIGHT)
68:  }

69:  cloneNode(Node nd, int i) {
70:      if nd = ⊥ then return
         // Assume atomic assignment
71:      datasetMemento[i] ← ⟨nd,nd.left,nd.right,nd.data,nd.color⟩
72:      add nd's memento color to the color set
73:  }
```

the owners of these nodes by "touching" them[1] (line 52) before advancing the operation to the RELEASE phase. This prevents other delayed processes from acquiring the nodes when the operation is not in the ACQUIRE phase (see Lemma 2(5) in the appendix).

To prevent a process from releasing nodes acquired in a later round, the operation adds the round number to any node it acquires (line 40); before a process releases a node, it verifies (line 30) that the round numbers of the owner and its own parameter are equal (see Lemma 2(4)).

Different operations extending the *Operation* class, refine the protocols for cloning and manipulating the data set, according to their specifications. Pseudocode 2 shows how the data set memento (the source node and both or one of its neighbors) is created. The applyChanges method changes the nodes according to the specification of the operation and maintains a legal coloring. Pseudocode 3 describes the implementation of these methods for the PushLeft and PopLeft operations.

The coloring of the nodes is kept legal by updating their color in the UPDATECOLOR method. The operation peeks at the colors of the neighboring nodes (lines 111–112) and sets a new color different from its neighbors (line 115). While it is possible that not both neighboring nodes are owned by the operation, it is guar-

[1] This is similar to the way an operation is invalidated before releasing its nodes in [4].

Pseudocode 3 CAS-Chromo: Detailed code for the apply-changes methods

```
74:  PushLeft::applyChanges() {
         // MIDDLE is the new node,
         // LEFT is the left anchor
75:      updateRight(MIDDLE,RIGHT)
76:      updateLeft(MIDDLE,LEFT)
             // the new node is now valid
77:      updateColor(MIDDLE)
78:      updateLeft(RIGHT,MIDDLE)
79:      updateRight(LEFT,MIDDLE)
80:  }

81:  PopLeft::applyChanges() {
         // MIDDLE is popped
         // LEFT is the left anchor
82:      if empty list then
83:          toReleaseEmptyState()
84:      if state.result = EMPTY then return
85:      setTempColor(RIGHT)
86:      updateRight(LEFT,RIGHT)
87:      updateLeft(RIGHT,LEFT)
88:      updateRight(MIDDLE,⊥)
             // the popped node is now invalid
89:      updateLeft(MIDDLE,⊥)
90:      updateColor(RIGHT)
91:  }

92:  updateRight(int i, int j) {
93:      nm_i ← get i-th node mementos
94:      nm_j ← get j-th node mementos
95:      nd ← nm_i.node
96:      newr ← nm_j.node
97:      rt ← nm_i.right
98:      CAS(nd.right, rt, ⟨newr,rt.aba+1⟩)
99:  }

100: updateLeft(int i, int j) {
101:     nm_i ← get i-th node mementos
102:     nm_j ← get j-th node mementos
103:     nd ← nm_i.node
104:     newl ← nm_j.node
105:     lft ← nm_i.left
106:     CAS(nd.left, lft, ⟨newl,lft.aba+1⟩)
107: }

108: updateColor(int i) {
109:     nm ← get i-th node memento
110:     nd ← nm.node
111:     lftc ← nd.left.node.color
112:     rtc ← nd.right.node.color
113:     newc ← color not in {lftc,rtc}
114:     clr ← nm.color
115:     CAS(nd.color, clr, ⟨newc,clr.aba+1⟩)
116: }

117: setTempColor(int i) {
118:     nm ← get i-th node memento
119:     nd ← nm.node
120:     clr ← nm.color
121:     CAS(nd.color, clr, ⟨c_0,clr.aba+1⟩)
122: }
```

anteed that their color does not change. We prove that nodes in the list are legally colored at all times (see Lemma 3).

Finally, the methods of Pseudocode 4 apply the state transitions of the operations, as depicted in the state transition diagram in Fig. 6.

3.2 DCAS-Chromo: A Doubly-Linked List Algorithm

We now describe the extensions needed to obtain DCAS-Chromo, which allows removals from the middle of the list. This may create long helping chains, as demonstrated in Fig. 8, which shows a long linked-list of nodes with alternating colors: red, yellow, red, yellow, ... Consider a set of concurrent operations, each of which is trying to remove a different yellow-colored node (at even positions), by acquiring the node and its two red-colored neighbors. If the two red-colored nodes are acquired one

Pseudocode 4 CAS-Chromo: Detailed code for state transitions

```
123: toInitState() {
124:   s ← state
125:   CAS(state,s,⟨s.seq+1,INIT,NULL⟩)
126: }

127: toAcqState() {
128:   s ← state
129:   if s.phase != INIT then return
130:   CAS(state,s,⟨s.seq,ACQUIRE,NULL⟩)
131: }

132: toApplyState(int seq) {
133:   s ← state
134:   if s != ⟨seq,ACQUIRE,NULL⟩ then
135:     return
136:   CAS(state,s,⟨seq,APPLY,SUCCESS⟩)
137: }

138: toReleaseState() {
139:   s ← state
140:   if s.phase != APPLY then return
       // linearization point—LP2
141:   CAS(state,s,⟨s.seq,RELEASE,s.result⟩)
142: }

143: toFinalState() {
144:   s ← state
145:   if s.phase != RELEASE then return
146:   CAS(state,s,⟨s.seq,FINAL,s.result⟩)
147: }
148: toFinalInvalidState() {
149:   s ← state
150:   if s.phase != INIT then return
       // linearization point—LP1
151:   CAS(state,s,⟨s.seq,FINAL,INVALID⟩)
152: }

153: toReleaseContentionState(int seq) {
154:   s ← state
155:   if s != ⟨seq,ACQUIRE,NULL⟩ then return
156:   CAS(state,s,⟨seq,RELEASE,CONTENTION⟩)
157: }

158: toReleaseEmptyState() {
159:   s ← state
160:   if s != ⟨seq,APPLY,SUCCESS⟩ then
161:     return
       // linearization point—LP3
162:   CAS(state,s,⟨seq,RELEASE,EMPTY⟩)
163: }
```

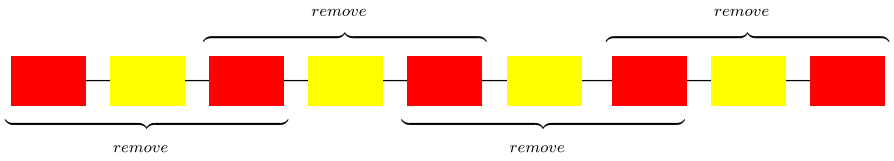


Fig. 8 Example: a symmetric scenario; each operation removes a different *yellow-colored* node (at even positions), and accesses the node and its two *red-colored* neighbors

at a time, in the same order, e.g., first the left neighbor, it is possible that an operation holds its left red node, and needs to help all operations to its right, in order to acquire the right left node.

One might suggest to extend the notion of a legal coloring and require that any triple of neighboring nodes is assigned different colors. This certainly allows to follow the color-based scheme, but how can we preserve this extended coloring property? In particular, when a node is removed, it is necessary to acquire *four* nodes in order to legally re-color the remaining three nodes; this requires to further extend the coloring property to any four consecutive nodes, which in turn requires to acquire *five* nodes and so on; this unlimited expansion of the coloring property seems inevitable.

Our way to break out of this vicious circle is to acquire equally-colored nodes *atomically*. An operation accesses at most three consecutive nodes, which are legally

colored, so at most two of them have the same color. We use DCAS to atomically acquire these nodes, e.g., the two red-colored nodes in the scenario of Fig. 8, and to break the symmetry.

Pseudocode 5 presents the code for the additional operation *Remove*, for removing a non-anchor valid node (the source node) from the list. The applyChanges method is very similar to the pop operations (removing nodes from the ends) except that it does not handle the case where the list is empty. The most important modification, relative to CAS-Chromo, is in the acquireColor method, which now uses DCAS when acquiring two nodes with the same color. That is, instead of acquiring the nodes one by one (Pseudocode 1, lines 39–40) DCAS acquires both nodes atomically (Pseudocode 5, lines 177–178).

4 Safety Proof: Linearizability

In this section, we prove that CAS-Chromo and DCAS-Chromo are linearizable (Theorem 1). The proof does not assume that DCAS is used, and thus it holds for both algorithms. Using DCAS is critical only for proving the locality properties of DCAS-Chromo, in Sect. 5.1.

The linearizability of both algorithms hinges on showing that the implementation follows the scheme. Namely, the executing processes preserve the correct phase transitions of the operation—acquiring, changing and releasing nodes—and take steps in accordance with the operations' phase. Most importantly, nodes in the data set are changed only while all of them are acquired. This is somewhat more complicated than in previous work [1, 4, 7, 24, 27], since the data set is dynamic.

The methods of Pseudocode 4 are the only way to make state transitions, and they imply that the state transitions of an operation follow the diagram in Fig. 6. Moreover, the round number is increased before every round. Hence, no operation makes a transition to the same state tuple more than once.

The following terminology is used in the proofs. The ACQUIRE phase of the r-th round is called the *r-th* ACQUIRE *phase*, and similarly for the other phases. The code implies that an operation is in the APPLY phase at most once (since all transitions from it are to a final state), in which case the operation completes and will not be re-invoked again; this is called the *last round*. Only the initiator generates the data set memento, once per round (line 11); the data set memento written in the r-th round is called the *r-th data set memento*; the data set memento of the last round is called the *last data set memento*.

Several executing processes can make the transitions to the APPLY and RELEASE phases concurrently, but other transitions are only made by the initiator. A transition to the APPLY phase only occurs once, in the last round; the method implementing a state transition to the RELEASE phase in case of contention takes as argument the round number, to ensure the transition occurs only if it is executed for the correct round.

The main invariants of the algorithm are as follows: nodes are only modified by the applyChanges, after the operation acquired all locks on them; modifications are done using a CAS, that verifies that the attribute being changed has not been modified

Pseudocode 5 Code changes for DCAS-Chromo

```
164: Remove::cloneDataset() {
165:     cloneNode(source, MIDDLE)
166:     left ← datasetMemento[MIDDLE].left.node
167:     cloneNode(left, LEFT)
168:     right ← datasetMemento[MIDDLE].right.node
169:     cloneNode(right, RIGHT)
170: }

171: Operation::acquireColor(Color c, int seq) {
172:     {nd_i} ← get all nodes with color c from the seq-th memento
173:     while true do
174:         {owner_i} ← get owners from {nd_i}
175:         checkNodes({nd_i},seq)
176:         if state != ⟨seq,ACQUIRE,NULL⟩ then return
          // atomically acquire two equally colored nodes
177:         if for each nd_i, owner_i = ⟨⊥,⊥,t_i⟩ then
178:             DCAS(nd_1.owner, nd_2.owner, owner_1, owner_2, ⟨self,seq,owner_1.aba+1⟩,
                    ⟨self,seq,owner_2.aba+1⟩)
179:         owner ← get owner from {nd_i} // check if succeeded or blocked
180:         if owner.op = self then return// acquired all c-colored nodes
181:         checkNodes({nd_i},seq)
182:         if state != ⟨seq,ACQUIRE,NULL⟩ then return
183:         if owner.op != ⊥ then // blocked by owner.op operation
184:             owner.op.help(owner.seq)// help blocking operation
185: }

186: Remove::applyChanges() {
       // the MIDDLE node in the memento is removed
187:     setTempColor(RIGHT)
188:     updateRight(LEFT,RIGHT)
189:     updateLeft(RIGHT,LEFT)
190:     updateRight(MIDDLE,⊥) // the removed node is now invalid
191:     updateLeft(MIDDLE,⊥)
192:     updateColor(RIGHT)
193: }
```

since it was cloned before the lock was acquired. These invariants protect the list from both concurrent changes by other contending operations, and from concurrent changes by helping processes.

The next two lemmas formalize the guarantees these techniques provide. Their proofs are deferred to Appendix A.

Lemma 1 *An operation op successfully applies changes only when it is in* APPLY *phase, and only to nodes in op's last data set memento.*

Lemma 2 *The following claims all hold for every operation op*:

1. *If op acquires a node t in the r-th round, then the r-th memento of t in op's data set is valid and t, excluding t's owner, has not changed after it was cloned by op in the r-th round.*
2. *op advances from the r-th* ACQUIRE *phase to the r-th* APPLY *phase only if all the nodes in its data set are consistent with the r-th data set memento, and are acquired by op.*
3. *op only applies changes to nodes that are acquired by it.*
4. *op releases nodes only when it is in* RELEASE *phase, and during its r-th* RELEASE *phase it only releases nodes it has acquired in the r-th round.*

Linearizability of CAS-Chromo and DCAS-Chromo immediately follows.

Theorem 1 CAS-Chromo *and* DCAS-Chromo *are linearizable.*

Proof We identify, for every operation, a *linearization point* inside its interval, so that the operation appears to occur atomically at this point. The linearization point of an operation op_i is either at the transition to state ⟨last,FINAL,INVALID⟩ (line 8), at the transition to state ⟨last,RELEASE,SUCCESS⟩ (line 24), or at the transition to state ⟨last,RELEASE,EMPTY⟩ (line 83); that is, when the CAS of the transition is applied successfully (line 151, marked LP1, line 141, marked LP2, or line 162, marked LP3, respectively). This is well defined, since only one of these points can occur.

In the first case, op_i discovers that the *source* node is invalid, since another operation op_j removes it. Before its transition to the FINAL phase, op_i helps op_j (line 7). By Lemma 1, the transition to the APPLY phase of op_j already occurred and op_i helps it to complete in case it has not completed yet. Thus, op_i need not apply its changes, and it is linearized after op_j.

In the second case, the pop operation op_i discovers that the list is empty (line 82) while owning both anchors. At this point no other operation can insert a node to the list, and the transition to the RELEASE phase with an EMPTY result is the linearization point of op_i.

In the third case, Lemma 2 (1) and (2) imply that when the transition to the APPLY phase occurs, all the nodes in the data set memento of op_i are valid, have not changed since they were cloned (except for their owners) and they are acquired by op_i. By Lemma 2(4), these nodes are not released while op_i is in the APPLY phase, which means, by Lemma 2(3), that no other operation changes these nodes and they are modified only by op_i during the execution of the APPLY phase. Finally, the pseudocode of the applyChanges methods and the use of CAS together with the ABA-prevention counter for any change applied to the nodes, guarantee that the executing processes of the operation preserves the specification of the corresponding doubly-linked list operations. □

5 CAS-Chromo and DCAS-Chromo Are Local Nonblocking

The legality of the coloring is now used to show that the algorithms are local nonblocking. Recall that a node is legally colored if its color differs from the colors of

its neighbors; the left anchor is legally colored if its color is different from its right neighbor, and the right anchor is legally colored if its color is different from its left neighbor.

5.1 DCAS-Chromo

It is simple to see that all operations change at most three consecutive nodes in the linked list and access at most four consecutive nodes, and that each operation only changes the color of a single node: insert and push operations change the color of the new node, and a pop and remove operations change the color of the right node in their data set. No operation changes the color of the left node in its data set. Since an operation changes a node only when owning it, this ensures that the colors of two neighboring nodes is not changed at the same time, even if concurrent operations access them.

By Theorem 1, we can assume that the changes are applied in isolation from other operations. Therefore, the proof of the next lemma, which appears in Appendix B, is merely a step-by-step sequential analysis of the applyChanges methods of the various operations.

Lemma 3 *All valid nodes are legally colored.*

An operation owns at most three consecutive nodes, and by the lemma, at most two of them have the same color, hence, DCAS suffices to acquire equally colored nodes in the operations' data set.

The proof that the algorithm is local nonblocking starts by showing that a process only helps operations within constant distance of the operation it is executing. The proof considers the number of help methods a process started executing but have not yet completed, and shows that this is a lower bound on the color already acquired by the operations the process helps to execute.

Lemma 4 *Consider process p that called $h > 0$* help *methods and completed $h' < h$ of them. If the last call is the* help(r_i) *method of an operation op_i, then during its r_i-th round, op_i only has to acquire nodes in its data set with color greater than or equal to $c_{h-h'}$.*

Proof The proof is by induction on $k = h - h'$. The base case is when $k = 1$, that is, $h' = h - 1$; in this case, p has only called the help method for its own operation. A node has color c_0 if it is a new node that is created by the operation during its initialization (omitted from the pseudo-code), or if it is the right node in a remove operation, which is colored c_0 during the APPLY phase. In both cases, the node is not acquired with color c_0. This means that only nodes with color greater than or equal to c_1 have to be acquired.

In the induction step, $k > 1$, implying that $h' < h - 1$. Hence, p called the help method for at least one operation other than itself and did not complete. Assume the penultimate help method called and not completed by p is for operation op_j. By the induction assumption, when the help method of op_j is called, op_j has to acquire

node t with color $c \geq c_{k-1}$. Process p reads t's owner (line 41), and discovers op_j failed to acquire t (line 42) and that t is consistent with its memento (lines 43–44), i.e., its color did not change. Then, p discovers that op_j is blocked by operation op_i (different than op_j) in its r_i-th round (line 45) and calls the last help method (line 46). Since equally colored nodes are acquired atomically, op_i acquired color c and only has to acquire nodes with color greater than or equal to c_k, and the lemma follows. □

Note that $h - h' - 1$ bounds from above the distance to operations that a process helps. Thus, Lemma 4 implies:

Corollary 1 *If op_i helps op_j, at distance d, then op_j only has to acquire nodes with color greater than c_d. In particular, op_j is in the 3-neighborhood of op_i, and if $d = 3$ then op_j completed the* ACQUIRE *phase.*

Afek et al. [1] define the next notion, capturing the locality of implementations in terms of memory contention:

Definition 3 An algorithm has *d-local contention* if two processes, p_1 and p_2, access the same memory location in operations op_1 and op_2, respectively, only if op_1 and op_2 are within distance d.

Theorem 2 (Local contention) DCAS-Chromo *has* 7*-local contention.*

Proof Two processes p_i and p_j access the same memory location if they help execute operations, op_k and op_l respectively, within distance one of each other. By Corollary 1, op_i is in the 3-neighborhood of op_k and op_j is in the 3-neighborhood of op_l. Thus, the distance between op_i and op_j is at most 7. □

Once an operation is in its APPLY phase, it is straightforward that it completes after one of its executing processes takes a constant number of steps. It remains to prove that if the operation is blocked outside the APPLY phase, then some "nearby" operation (in a sense made precise by Lemma 5) completes. We do so by considering executions of the loop of acquireColor(c, r) method (lines 35–46), called a *c-acquiring iteration*. Lemma 5 below shows that in every acquiring iteration of an executing process, whether successful or not, some "nearby" operation makes progress.

Fix an arbitrary operation op_b initiated by process p_b; progress in the neighborhood of op_b is tracked by three counters:

- The *completed operations counter*, denoted co, initially 0, is increased whenever an operation in the 4-neighborhood of op_b completes.
- The *color counter*, denoted cl, holds the color of the last acquiring iteration that the initiator of op_b executed.
- The *changes counter*, denoted ch, initially 0, is increased whenever an operation in the 5-neighborhood of op_b changes an item in its data set.

The values of the counters in a configuration C are denoted $co(C)$, $cl(C)$, $ch(C)$; co and ch are nondecreasing, but cl is not necessarily monotone.

Assume that process p_b takes an infinite number of steps, executing infinitely many acquiring iterations, without completing op_b. Let the configurations at the start of the acquiring iterations of p_b be denoted $C_0, C_1, \ldots, C_t, \ldots$, in the order they occur. The next lemma argues that each configuration C_t is a milestone in the progress of the operations in op_b's neighborhood.

Lemma 5 *For every $t > 0$, either $co(C_t) > co(C_{t+1})$, or $cl(C_t) > cl(C_{t+1})$, or $ch(C_t) > ch(C_{t+1})$.*

Proof Assume that process p_b starts a c-acquiring iteration at C_{t-1}, during an acquireColor(c, r) method of op, the j-th operation of p_i, and a c'-acquiring iteration at C_t, during an acquireColor(c', r') method of op', the j'-th operation of $p_{i'}$. By Corollary 1, op and op' are in the 3-neighborhood of op_b.

First, assume that $i \neq i'$, i.e., the operations have different initiators.

If op completes before the second iteration, then $co(C_{t-1}) < co(C_t)$. Otherwise, the first acquiring iteration of op failed. If the failure is due to contention, then some operation op_l at distance one from op applied a change to its data set; since op_l is in the 4-neighborhood of op_b, $ch(C_{t-1}) < ch(C_t)$. Otherwise, the first acquiring iteration of op failed since op' blocked it. That is, op fails to acquire a node t with color c since it is already owned by op', and then p_b helps op' and executes the second c'-acquiring iteration. Since op' atomically acquires all the nodes with color c, $c < c'$, we get $cl(C_{t-1}) < cl(C_t)$.

Next, assume that $i = i'$, i.e., the operations have the same initiator, and therefore, $j \leq j'$.

If $j < j'$, p_i completed its j-th operation and $co(C_{t-1}) < co(C_t)$.

Otherwise, $j = j'$, i.e., both acquiring iterations are of the same operation. The round number of the acquiring iterations is monotonically increasing, thus, $r \leq r'$. If $r < r'$, then op is re-invoked before the second acquiring iteration, due to contention in the first iteration. Thus, in the first iteration some operation op_l at distance one from op applied a change to its data set that failed op. The operation op_l is in the 4-neighborhood of op_b and $ch(C_{t-1}) < ch(C_t)$.

Otherwise, $r = r'$, i.e., both acquiring iterations are of the same round. The colors p_b is acquiring in the same round of the same operation are nondecreasing, thus, $c \leq c'$. If $c < c'$ then $cl(C_{t-1}) < cl(C_t)$.

Finally, we are left with the case that $i = i'$, $j = j'$, $r = r'$, and $c = c'$, that is, two consecutive c-acquiring iterations in the same round of the same operation. The process executing the acquiring iterations, p_b, fails to acquire some node t with color c in the first iteration. Then, without helping any other operation (since the node is already released), p_b retries the acquiring iteration. The process p_b fails to acquire t in the first iteration since another operation op_l, in the 1-neighborhood of op, holds t. The operation op_l is in the 4-neighborhood of op_b. If op_l releases t after it completes, then $co(C_{t-1}) < co(C_t)$. Otherwise, the node is released since op_l discovers that a node it is trying to acquire, t', is inconsistent with its memento. Thus, another operation op_k in the 1-neighborhood of op_l changed t' after op_l generated the memento of t'. Since op_k is in the 5-neighborhood of op_b, we get $ch(C_{t-1}) < ch(C_t)$. □

Theorem 3 DCAS-Chromo *is a* 5*-local nonblocking implementation of a doubly-linked list.*

Proof Consider the initiator p_b of an operation op_b. By Lemma 5, at least one counter increases with each acquiring iteration of p_b. Since the color counter is at most 3, after at most three consecutive acquiring iterations, some counter other than the color counter must increase. If the completed operations counter increases, then some pending operation in the 4-neighborhood of op_b completes. Otherwise, the changes counter increases. Once it is in the APPLY phase, an operation completes within a constant number of changes. Thus, after p_b executes a number of acquiring iterations that is linear in the number of operations in the 5-neighborhood of op_b, some pending operation in the 5-neighborhood of op_b completes. Since there is a finite number of processes, it follows that after p_b takes an finite number of steps, some operation in the 5-neighborhood of op_b completes. □

5.2 CAS-Chromo

CAS-Chromo does not support removals from the middle of the linked list, and hence only *pop* operations acquire three consecutive nodes, two of which may have the same color. Since operations acquire equally colored nodes by their order in the list from left to right, whenever a process calls a new help method it helps a new operation. Thus, the number of *pop* operations a process started helping and did not complete is at most two, and helping cycles are avoided.

We revise Lemma 4, Corollary 1, and Lemma 5.

Lemma 4′ *Consider process p that called $h > 0$* help *methods and completed $h' < h$ of them, such that from the uncompleted* help *methods excluding the last (altogether $h - h' - 1$ methods), ℓ are of* pop *operations. If the last call is the* help(r_i) *method of an operation op_i, then during its r_i-th round, op_i only has to acquire nodes in its data set with color greater than or equal to $c_{h-h'-\ell}$.*

Proof The proof is by induction on $k = h - h'$. The base case, $k = 1$, follows by arguments similar to the base case in the proof of Lemma 4.

In the induction step, $k > 1$, implying that $h' < h - 1$. Hence, p called the help method for at least one operation other than itself and did not complete. Assume the penultimate help method called and not completed by p is the help method of operation op_j.

We first assume that op_j is a pop operation. By the induction assumption, when the help method of op_j is called, op_j only has to acquire nodes with color greater than or equal to $c_{k-1-(\ell-1)} = c_{k-\ell}$. Process p helps op_j to acquire a node t with color $c \geq c_{k-\ell}$. We review p's steps while helping op_j to show that op_i acquired color c: p reads t's owner (line 41), and discovers op_j failed to acquire t (otherwise it returns in line 42) and that t is consistent with its memento (line 43), i.e., its color is still c. Then p discovers that op_j is blocked by operation op_i in its r_i-th round (line 45) and calls the last help method (line 46). The operation op_i acquired t with color c in its r_i-th round; op_i only has to acquire nodes with color greater than or equal to $c_{k-\ell}$, and the lemma holds.

If op_j is not a pop operation, then the induction assumption implies that when the help method of op_j is called, op_j only has to acquire nodes with color greater than or equal to $c_{k-1-\ell}$. Process p helps op_j to acquire the next node t. Since op_j is not a pop operation t has color $c \geq c_{k-\ell}$. Process p takes the same steps as in the previous case, to find that op_j is blocked by op_i in its r_i-th round, then it calls the last help method. The operation op_i, acquired t with color c in its r_i-th round; op_i only has to acquire nodes with color greater than or equal to $c_{k-\ell}$, and the lemma holds. □

Corollary 1′ *If op_i helps op_j, at distance d, then op_j already acquired a node with color greater or equal to c_{d-2}. In particular, op_j is in the 5-neighborhood of op_i, and if $d = 5$ then op_j completed the* ACQUIRE *phase.*

As in the proof of Theorem 2 for DCAS-Chromo, this implies that CAS-Chromo has 11-local contention; we next prove it is 7-local nonblocking.

The proof of Lemma 5 uses the fact that DCAS-Chromo atomically acquires all nodes with the same color. CAS-Chromo, however, acquires nodes with the same color one by one, from left to right. To cover this case, we use a *monochromatic chain counter*, denoted mc, to track the number of consecutive acquiring iterations that the initiator of an operation op_b executes with the same color; mc is reset whenever a different color is acquired. Furthermore, the neighborhoods for which the other counters are defined are extended, so that co is defined for the 6-neighborhood of op_b, and ch is defined for the 7-neighborhood of op_b.

Lemma 5′ *For every $t > 0$, either $co(C_t) > co(C_{t+1})$, or $cl(C_t) > cl(C_{t+1})$, or $mc(C_t) > mc(C_{t+1})$, or $ch(C_t) > ch(C_{t+1})$.*

Proof Assume that process p_b starts a c-acquiring iteration at C_{t-1}, during an acquireColor(c, r) method of op, the j-th operation of p_i, and a c'-acquiring iteration at C_t, during an acquireColor(c', r') method of op', the j'-th operation of $p_{i'}$. By Corollary 1′, op and op' are in the 5-neighborhood of op_b.

First, assume that $i \neq i'$, i.e., the operations have different initiators.

If op completes before the second iteration, then $co(C_{t-1}) < co(C_t)$. Otherwise, the first acquiring iteration of op failed. If the failure is due to contention, then some operation op_l at distance one from op applied a change to its data set; since op_l is in the 6-neighborhood of op_b, $ch(C_{t-1}) < ch(C_t)$. Otherwise, the first acquiring iteration of op failed since op' blocked it. That is, op fails to acquire a node t with color c since it is already owned by op', and then p_b helps op' and executes the second c'-acquiring iteration. This is the main point where the proof differs from the original one. If $c < c'$ then $cl(C_{t-1}) < cl(C_t)$, otherwise, $mc(C_{t-1}) < mc(C_t)$.

Next, assume that $i = i'$, i.e., the operations have the same initiator, and therefore, $j \leq j'$.

If $j < j'$, p_i completed its j-th operation and $co(C_{t-1}) < co(C_t)$.

Otherwise, $j = j'$, i.e., both acquiring iterations are of the same operation. The round number of the acquiring iterations is monotonically increasing, thus, $r \leq r'$. If $r < r'$, then op is re-invoked before the second acquiring iteration, due to contention in the first iteration. Thus, in the first iteration some operation op_l at distance one

from op applied a change to its data set that failed op. The operation op_l is in the 6-neighborhood of op_b and $ch(C_{t-1}) < ch(C_t)$.

Otherwise, $r = r'$, i.e., both acquiring iterations are of the same round. The colors p_b is acquiring in the same round of the same operation are nondecreasing, thus, $c \leq c'$. If $c < c'$ then $cl(C_{t-1}) < cl(C_t)$.

Finally, we are left with the case that $i = i'$, $j = j'$, $r = r'$, and $c = c'$, that is, two consecutive c-acquiring iterations in the same round of the same operation. The process executing the acquiring iterations, p_b, fails to acquire some node t with color c in the first iteration. Then, without helping any other operation (since the node is already released), p_b retries the acquiring iteration. The process p_b fails to acquire t in the first iteration since another operation op_l, in the 1-neighborhood of op, holds t. The operation op_l is in the 6-neighborhood of op_b. If op_l releases t after it completes, then $co(C_{t-1}) < co(C_t)$. Otherwise, the node is released since op_l discovers that a node it is trying to acquire, t', is inconsistent with its memento. Thus, another operation op_k in the 1-neighborhood of op_l changed t' after op_l generated the memento of t'. Since op_k is in the 7-neighborhood of op_b, we get $ch(C_{t-1}) < ch(C_t)$. □

Only pop operations in CAS-Chromo have three nodes in their data set and can be part of a monochromatic chain. Since equally-colored nodes are acquired from left to right, pop operations cannot form monochromatic *cycles*. Therefore, the length of a monochromatic chains is at most two, and it can be shown in a manner similar to the proof of Theorem 3, for DCAS-Chromo, that if the operation never ends, other operations in its 7-neighborhood complete.

Theorem 4 CAS-Chromo *is a* 7*-local nonblocking implementation of a doubly-linked list, allowing removals only at the ends.*

An implementation of a *deque* restricts insertions and removals to the ends. In this case, each operation has an anchor in its data set. This bounds the length of a path in the conflict graph to 3, that is, two operations are connected only if they are in the 3-neighborhood of each another.

Theorem 5 CAS-Chromo *is an implementation of a deque that has* 3*-local contention and is* 3*-local nonblocking.*

6 Related Work

Two research threads are related to our results: generic schemes based on acquiring ownership, and specialized algorithms for linked-list data structures.

6.1 Generic Ownership-Based Schemes

Several papers [7, 24, 27] systematically derive implementations of concurrent data structures, by acquiring data items one-by-one and applying changes after all items are acquired. While acquiring items, an operation may hold one or more items while

waiting for another operation to release another item; this can create long hold-and-wait chains.

The operations may help *recursively*, namely, a process helps another process to help a third process and so on, possibly causing long *helping chains*. For example, assume the nodes in Fig. 3 are acquired from left to right. Consider an execution α in which op_2, op_3 and op_4 concurrently acquire their left-most nodes successfully, and then op_1 tries to acquire its nodes while the other operations are delayed. Since m_2 is owned by op_2, op_1 has to help op_2; since m_4 is owned by op_3, op_1 has to help op_3; and since m_5 is owned by op_4, op_1 has to help op_4. Thus op_1 is delayed by a chain of operations, with intersecting data sets. In general, op_1 can be delayed by any operation within a finite distance from it, implying that the implementation is not local.

In some implementations [24], an operation helps only an immediate neighbor. Nevertheless, the number of steps a process performs depends on the length of the longest path connected to this operation in the conflict graph. Consider again an execution that starts with op_2, op_3 and op_4 acquiring their low-address nodes successfully, then op_1 fails to acquire m_2, op_2 fails to acquire m_4, and op_3 fails to acquire m_5; each operation then helps its (immediate) neighbor. Prior to helping, op_2 and op_3 release their nodes and stop taking steps, thus op_1 and op_2 discover their help is unnecessary. Assume that op_4 completes, and again op_1, op_2 and op_3 try to acquire their data sets. It is possible that op_2 and op_3 acquire their low-address nodes, and op_1 tries, in vain, to help op_2, which releases its nodes due to op_3, etc. When the path of overlapping operations gets longer, op_1 futilely helps op_2 more times.

The *color-based* scheme for implementing *binary* operations [4] bounds the length of helping chains by coloring the data items with ordered colors. An operation starts by coloring the items it is going to access with a constant number of colors, so that neighboring items have different colors, and then acquires data items in an increasing order of colors. In this scheme, op helps op' only if op' already owns a higher color. As in our algorithms, the length of helping chains is bounded by the number of colors, and an operation helps only operations at constant distance.

Afek et al. [1] extend this scheme to arbitrary k-ary operations. In addition to local contention (Definition 3), they also define an implementation to have *d-local step complexity*, if the step complexity of an operation depends only on the number of operations within distance d of it.

Both schemes [1, 4] must color the items when an operation starts, since the data set is arbitrary. This requires information about operations (and their data sets) at non-constant distance, leading to $O(\log^* n)$-local step complexity and contention, and a more complicated implementation. Since the data sets of linked-list operations is predicable, we can avoid the cost of initial coloring with a built-in coloring, yielding $O(1)$-local step complexity.

6.2 Previous Linked-List Algorithms

Several papers proposed implementations of dynamic linked list data structures (see Table 1).

Harris [14] used CAS to implement a singly-linked list, with insertions and removals anywhere; however, in this algorithm, a process can access a node previously

Table 1 Linked list algorithms; *interference* is the distance between operations that delay each other

Algorithm	Insertions	Removals	Uses dcas	Interference	Comments
Harris [14]	anywhere	anywhere	no	any pair	singly-linked list
Greenwald [12]	anywhere	anywhere	yes	any pair	
Michael [20]	anywhere	anywhere	no	any pair	singly-linked list
Sundell and Tsigas [25]	anywhere	anywhere	no	any pair	
DCAS-Chromo	anywhere	anywhere	yes	distance ≤ 5	
Greenwald [11]	ends	ends	yes	opposite ends	
Agesen et al. [2]	ends	ends	yes	distance $= 1$	
Michael [21]	ends	ends	no	opposite ends	
Herlihy et al. [17]	ends	ends	no	distance $= 1$	obstruction free, array-based
Sundell and Tsigas [26]	ends	ends	no	distance ≤ 2	
CAS-Chromo	ends	ends	no	distance ≤ 3	
CAS-Chromo	anywhere	ends	no	distance ≤ 7	

removed from the linked list, possibly yielding an unbounded chain of uncollected removed nodes. Michael [20] fixed these memory management issues. Elsewhere [21], Michael proposed an implementation of a deque; in this algorithm, a *anchor* single word holds the head and tail pointers, causing all operations to interfere with each other, and making the implementation inherently sequential.

Sundell and Tsigas [26] avoid the use of a single anchor, allowing operations on the two ends to proceed concurrently. An extension of this algorithm allows insertions and removals in the middle of the list [25]; in this algorithm, a long path of overlapping removals may cause interference among distant operations; moreover, during intermediate states, there can be a consecutive sequence of inconsistent backward links, causing part of the list to behave as singly-linked.

An *obstruction-free* deque, providing a weaker progress property, was proposed by Herlihy et al. [17]; besides blocking when there is even a little contention, this array-based implementation bounds the deque's size.

Greenwald [11, 12] uses DCAS to simplify the design of many data structures. His implementations of deques, singly-linked and doubly-linked lists synchronize via a single designated memory location, resulting in a strictly sequential execution. Agesen et al. [2] present the first DCAS-based, dynamically-sized deque implementation, supporting concurrent access to both ends of the deque, with 1-local step complexity; this algorithm does not allow operations in the middle of the linked list. SNARK [8] is an attempt for further improvement that uses only a single DCAS primitive per operation in the best case, instead of two. Unfortunately, SNARK is incorrect and the corrected version allows removed nodes to be accessed from within the deque, thus preventing the garbage collector from reclaiming long chains of unused nodes [9]. The authors suggest that primitives more powerful than DCAS, e.g., 3CAS, are needed in order to obtain simple and efficient implementations of data structures guaranteeing that some operation makes progress at any time.

DCAS-Chromo (as well as other algorithms [5]) indicate that DCAS provides significant leverage for achieving these goals, beyond what is offered by CAS. Indeed,

unlike CAS-based implementations, our implementations do not allow chains of removed nodes to be accessed from within the linked-list. While we have not tested their performance, we remark that in our algorithms, an operation completes within $O(1)$ steps if it is running solo in an execution suffix, i.e., it has constant *obstruction-free step complexity* [10].

7 Discussion

This paper presents a new approach for designing high-throughput implementations of linked list data structures. We show a DCAS-based implementation of insertions and removals in a doubly-linked list; for a deque and priority queues, where nodes are removed only from the ends, the implementation uses only CAS. These implementations are intended as a proof-of-concept and require further optimizations to make them more practical; it is also necessary to implement a *search* operation in order to support the full functionality of priority queues and lists. Finally, it is interesting to explore other applications of our scheme, e.g., for tree-based data structures.

DCAS-Chromo uses DCAS, which is seldom provided in hardware, but DCAS is an ideal candidate to be supported by *hardware transactional memory* [19, 22], being a short transaction with small static data set. Alternatively, DCAS can be simulated in software from CAS [4, 10], or by applying a simple randomized algorithm [13]. In particular, using the highly-concurrent implementation of Attiya and Dagan [4], which is $O(\log^* n)$-local nonblocking, yields an implementation that is $O(\log^* n)$-local nonblocking, using only CAS.

Acknowledgements We thank David Hay, Danny Hendler, Gadi Taubenfeld and the anonymous referees for helpful comments. This research was supported by the *Israel Science Foundation* (grants 1344/06 and 1227/10).

Appendix A: Proofs of Lemmas 1 and 2

Lemma 1 *An operation op successfully applies changes only when it is in* APPLY *phase, and only to nodes in op's last data set memento.*

Proof An executing process of *op* that calls applyChanges first verifies that the operation is in the APPLY phase (line 22), implying that this is the last round of the operation, and it holds the last data set memento. Since an operation changes only nodes in its data set memento, executing processes apply the same changes on the same nodes (from the last data set memento).

Let p_j be the process that advances *op* from APPLY phase to RELEASE phase (line 24). Before changing the state, p_j executes applyChanges (line 23), thus CAS is applied at least once to each of the attributes based on values in the node mementos (see the code in Pseudocode 3), prior to the state transition. Once a CAS is applied successfully the attribute is inconsistent with its memento since at least the ABA-prevention counter has changed. If after the transition another process p_i applies CAS to some attribute while applying *op*'s changes, p_i's CAS fails. □

In order to prove Lemma 2(1–4), it is helpful to inductively carry two additional claims (5 and 6).

Lemma 2′ (extended) *The following claims all hold for every operation op*:

1. *If op acquires a node t in the r-th round, then the r-th memento of t in op's data set is valid and t, excluding t's owner, has not changed after it was cloned by op in the r-th round.*
2. *op advances from the r-th* ACQUIRE *phase to the r-th* APPLY *phase only if all the nodes in its data set are consistent with the r-th data set memento, and are acquired by op.*
3. *op only applies changes to nodes that are acquired by it.*
4. *op releases nodes only when it is in* RELEASE *phase, and during its r-th* RELEASE *phase it only releases nodes it has acquired in the r-th round.*
5. *op acquires nodes only when it is in* ACQUIRE *phase, and during its r-th* ACQUIRE *phase it only acquires nodes that are in its r-th data set memento.*
6. *When an executing process of op returns from* acquireColor(c, r), *either all nodes with color c in the r-th data set memento are acquired by op, or op is not in the r-th* ACQUIRE *phase.*

Proof The claims are proved simultaneously by induction on the execution length. In the base case, the empty execution, all claims vacuously hold. In the induction step, we consider each claim:

1. If the r-th memento of a node t in op's data set is invalid no executing process tries to acquire t in the r-th round (line 37, and line 52).

   Now, assume t has changed after op cloned it in the r-th round, and that p_i, an executing process of op calls acquireColor(c, r), where c is the color of t. If the change occurs before p_i verifies t is consistent with its memento (line 37), then p_i does not try to acquire t. Otherwise the change occurs after verifying the consistency, and in particular after p_i reads the content of t's owner (line 36). By Lemma 1, the change is applied by an operation op' in its APPLY phase. By (3), t is owned by op' when applying the change. If op' acquired t before p_i reads the content of t's owner, then by (4) it is not released until after op' applied the changes, i.e., until after p_i reads the owner, then p_i does not try to acquire t. Otherwise, op' acquired t after p_i reads t's owner, in this case, p_i fails acquiring t's owner, since the ABA-prevention counter has changed.
2. A transition of op from the r-th ACQUIRE phase to the r-th APPLY phase by an executing process occurs only after the executing process calls acquireColor(c, r) with all colors from the r-th color set while op is in the r-th ACQUIRE phase. By (6), when returning from each such round, all relevant nodes are acquired by op, and by (4), they were not released since then. As the set of colors includes all nodes in the r-th data set memento, they are all owned by op while the transition occurs. By (1), no node in the r-th data set memento is changed before op advances to the APPLY phase. So, when the transition occurs all nodes in the r-th data set memento are acquired by op, and they are consistent with their mementos.
3. By Lemma 1, op only applies changes while it is in APPLY phase and only to nodes that are in the last data set memento. By (2), when the transition to APPLY

phase occurs all nodes in the r-th (last) data set memento are acquired by op, and by (4), it does not release the nodes while it is in the APPLY phase, implying that op changes a node only while owning it.

4. The transition of op from the r-th RELEASE phase to the r-th FINAL phase (line 14) by the initiator of op, p, occurs after p calls the releaseDataset method (line 25), to release all nodes in the r-th data set memento (line 28). All processes executing the r-th round try to release the same nodes from the r-th data set memento, since by (5), op only acquires nodes from the r-th data set memento in the r-th round. When a process p_i releases op's data set while executing the r-th round of op, it reads the owner on a node t (line 29), and tries to release t (line 31) after verifying that op is in the RELEASE phase and t is owned by op in its r-th round (line 30). It can be easily verified from the code that after p completes the releaseDataset method all the nodes that were acquired by op are released. Thus, if p_i tries to release the nodes after the transition occurs, the CAS fails since the ABA-prevention counter has changed.
5. Consider an executing process p_i executing the r-th round of op. First p_i reads t from op's r-th data set memento (line 34); then p_i reads t's owner (line 36); verifies that t's memento is valid (line 37); and that op is in the r-th ACQUIRE phase (line 38). Let p_j be the executing process that advances op from the r-th ACQUIRE phase either to the r-th APPLY phase or to the r-th RELEASE phase. Assume the transition occurs after p_i verifies the phase, and specifically after it reads t's owner, but before p_i tries to acquire t (line 40). If p_j advances op to the r-th APPLY phase, by (2) all nodes in the r-th data set memento, including t, are acquired by op when p_j makes the transition. Thus either p_i discovers that t is owned by op or it fails acquiring t's owner, since at least the ABA-prevention counter has changed.

   Otherwise, p_j advances to the r-th RELEASE phase since it discovers that some node t' in the r-th data set memento is inconsistent with its memento (line 37 or line 43). There are three cases depending on the order between the colors of the nodes:
   (i) t' and t have the same color. Before the transition occurs, p_j violates the owners of both nodes by "touching" their ABA-prevention counter (line 52). By (1), since t changed while op is in the r-th ACQUIRE phase, p_i cannot successfully acquire it in this round.
   (ii) t' has lower color than t. Thus, p_i executes acquireColor(c', r), where c' is the color of t', before trying to acquire t. By (6), t' was acquired by op and by (1), it has not changed while op is in the r-th ACQUIRE phase, which contradicts the assumption that p_j discovers it is inconsistent with its memento.
   (iii) t' has higher color than t. Thus, p_j executes acquireColor(c, r), where c is the color of t, before discovering the change in t'. By (6), t was acquired by op. Thus either p_i discovers that t is owned by op or it fails acquiring t, since at least the ABA-prevention counter has changed.
6. An executing process of op that executes acquireColor(c, r) returns from the method in one of three cases: Two cases are when it recognizes a change in the state (line 38 or line 44), and it is evident by the state diagram that when returning from the method, op is no longer in the r-th ACQUIRE phase. The third case is

after verifying all the nodes with color c in the r-th data set memento are acquired by op (line 42). Now, if when returning from the method some of the nodes are not owned by op, then, by (4), they are released by the operation that acquired them, while it is in RELEASE phase, so this case also satisfies the condition. □

Appendix B: Proof of Lemma 3

Recall that a node is valid if it is an anchor or both its left and right links are not null.

The proof is by induction on the execution order. In the base case, the linked list is empty: the left anchor is colored c_1 and the right anchor is colored c_3, and hence, they are legally colored.

Induction step: a node can become illegally colored only when some operation applies its changes to the node or one of its neighbors. By Lemma 2(3), an operation changes a node only if it owns it. This implies that no node is inserted or removed immediately to the left or to the right of an operation data set while the operation applies its changes. Moreover, by the above observation a pop and remove operations only change the color of the right node in the data set and an insert or push operations only change the color of the new, i.e., middle, node in the data set. Thus, we can derive the next claim:

Claim *An operation changes a node's color only if it holds the node and its left neighbor.*

We analyze every step in the APPLY phase of an operation, and we show that after each such step the node that was changed is still legally colored. Consider first the *PushLeft* operation presented in Fig. 9. The data set of the operation, op_1, is the new node (m), the left anchor (LA), and its right neighbor (m_1). While op_1 is applying its changes, other operations neither remove nor insert nodes to the right of the right neighbor node, and also do not change the colors of the nodes in the data set. The claim implies that other operations do not change the color of an additional right neighbor (m_2). For *PushRight* or insert operations, the induction assumption also implies that a left neighbor is legally colored even if its color is changed by another operation. It remains to show, by inspecting the code, that the changes applied by the operation keep the nodes legally colored.

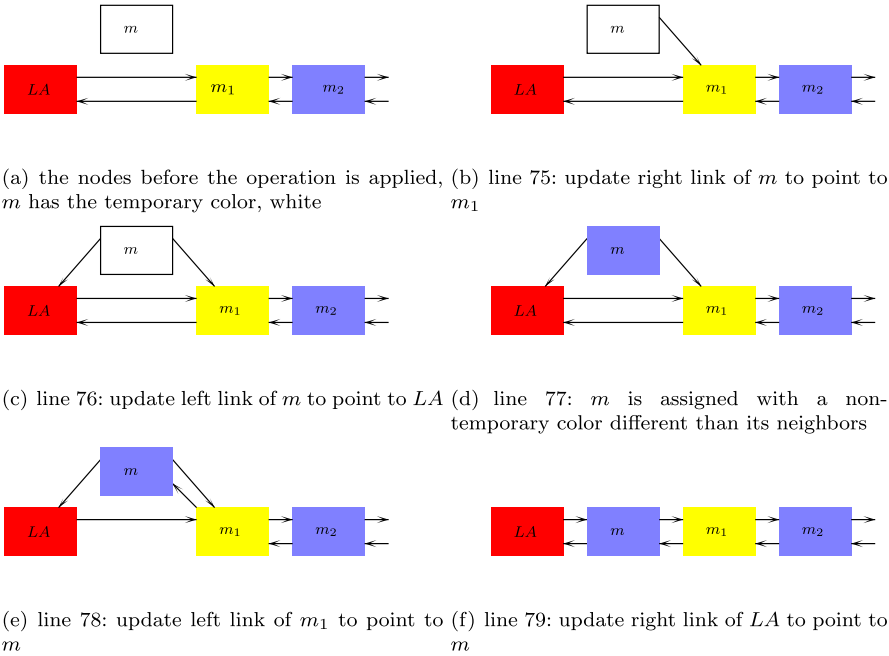
Line 75 update right link of the new node: the new node is not yet valid (Fig. 9(b));

Line 76 update left link of the new node: the new node is valid and it is legally colored (Fig. 9(c));

Line 77 the new node is assigned with a non-temporary color different than its neighbors and the new node is legally colored (Fig. 9(d));

Line 78 update left link of the right neighbor (m_1): the right neighbor has color different than the colors of the new node and the right neighbor (m_2), and thus it is legally colored (Fig. 9(e)).

Line 79 update right link of the left anchor (or the source node in the case of an *insertRight* operation): the left anchor has color different than the color of the new node (in the case of an *insertRight* operation the source node also has color different than the color of the left neighbor), and thus it is legally colored (Fig. 9(f));



(a) the nodes before the operation is applied, m has the temporary color, white

(b) line 75: update right link of m to point to m_1

(c) line 76: update left link of m to point to LA

(d) line 77: m is assigned with a non-temporary color different than its neighbors

(e) line 78: update left link of m_1 to point to m

(f) line 79: update right link of LA to point to m

Fig. 9 Example: the applyChanges method for *PushLeft* operation (Pseudocode 3). Given the nodes LA, m_1, m_2 from Fig. 3, op_1 pushes a new node, m, into the left side of the list

We next analyze the *PopLeft* operation (see Fig. 10). The data set of the operation, op_2, is the left anchor (LA) and its right neighbors (m_1 and m_2). While op_2 is applying its changes, other operations neither remove nor insert nodes to the right of the right neighbor, and also do not change the colors of the nodes in the data set. The claim implies that other operations do not change the color of an additional right neighbor (m_3). For a *PopRight* operation, the induction assumption implies that the left neighbor is legally colored even if its color is changed by another operation. We show again that the operation's changes keep the nodes legally colored:

Line 85 the right neighbor (m_2) is assigned with the temporary color: the right neighbor is legally colored, since the subject node (m_1) and the right neighbor (m_3) have colors different than the temporary color (Fig. 10(b));

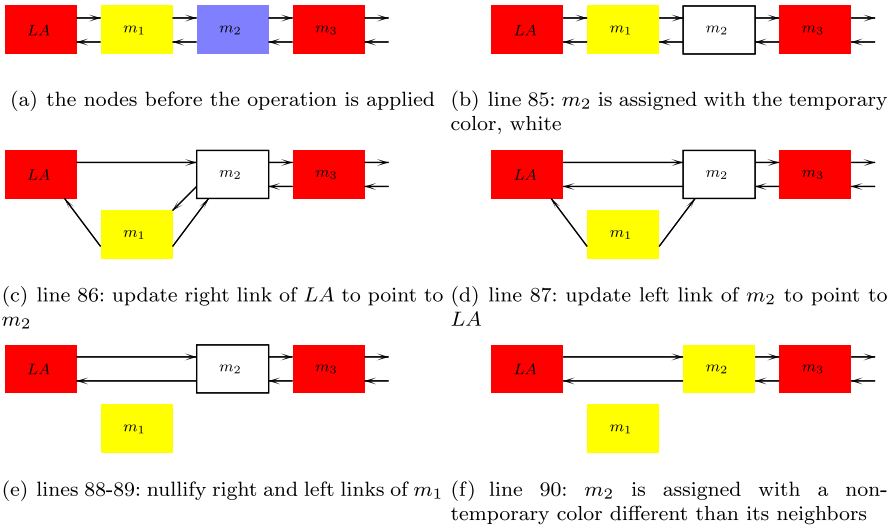
Line 86 update right link of the left anchor (m_2): the left anchor has a non-temporary color, and thus it is legally colored (Fig. 10(c));

Line 87 update left link of the right neighbor: the right neighbor is legally colored, since the left anchor and the adjacent node to the right have colors different than the temporary color (Fig. 10(d));

Line 88 set right link of the subject node to null: the subject node is now invalid (Fig. 10(e));

Line 89 set left link of the subject node to null: the subject node is invalid (Fig. 10(e));

Line 90 the right neighbor is assigned with a non-temporary color different than its neighbors, and thus it is legally colored (Fig. 10(f)).



(a) the nodes before the operation is applied

(b) line 85: m_2 is assigned with the temporary color, white

(c) line 86: update right link of LA to point to m_2

(d) line 87: update left link of m_2 to point to LA

(e) lines 88-89: nullify right and left links of m_1

(f) line 90: m_2 is assigned with a non-temporary color different than its neighbors

Fig. 10 Example: the applyChanges method for *PopLeft* operation (Pseudocode 3). Given the nodes LA, m_1, m_2, m_3 from Fig. 3, op_2 pops the node m_1

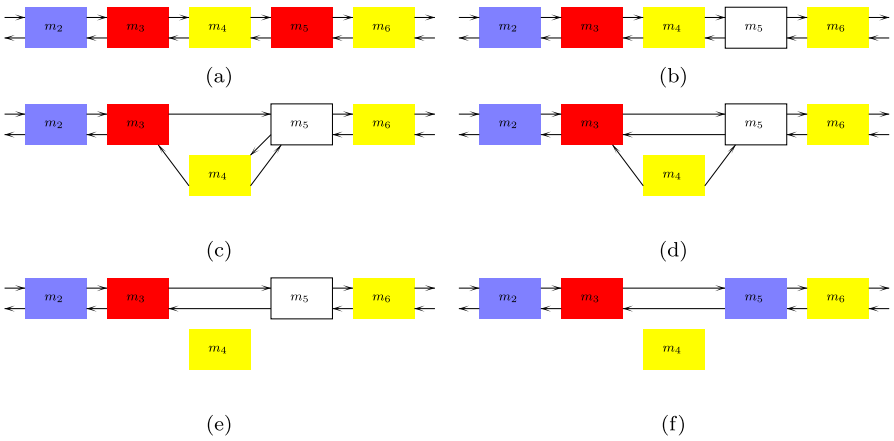


Fig. 11 Example: an execution of a *Remove* operation—op_4 from Fig. 3

It remains to show that the *remove* operation keeps the coloring legal (see Fig. 11). The remove operation manipulates its data set in a manner similar to the pop operation. We analyze the *Remove* operation, op_4, presented earlier in Fig. 3. Figure 11(a) presents the nodes m_2, m_3, m_4, m_5, m_6 from Fig. 3, op_4 removes the source node m_4. The data set of the operation is the source node and its neighbors (m_3 and m_5). During the ACQUIRE phase, op_4 first acquires m_4, and then atomically acquires m_3 and m_5, which are equally colored. While op_4 is applying its changes, other operations neither remove nor insert nodes to the left of the left neighbor and to the right of the right neighbor, and also do not change the colors of the nodes in the data set.

The claim implies that other operations do not change the color of an additional right neighbor (m_6). Moreover, the left neighbor (m_2) is legally colored even if its color is changed by another operation, while op_4 is applying its changes. The detailed description follows along the same lines as the *PopLeft* operation.

References

1. Afek, Y., Merritt, M., Taubenfeld, G., Touitou, D.: Disentangling multi-object operations. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 111–120 (1997)
2. Agesen, O., Detlefs, D.L., Flood, C.H., Garthwaite, A.T., Martin, P.A., Shavit, N., Steele, G.L.: DCAS-based concurrent deques. Theory Comput. Syst. **35**(3), 349–386 (2002)
3. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Syst. **34**(2), 115–144 (2001)
4. Attiya, H., Dagan, E.: Improved implementations of binary universal operations. J. ACM **48**(5), 1013–1037 (2001)
5. Attiya, H., Hillel, E.: Highly concurrent multi-word synchronization. Theor. Comput. Sci. **412**(12–14), 1243–1262 (2011)
6. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. Wiley-Interscience, New York (2004)
7. Barnes, G.: A method for implementing lock-free shared-data structures. In: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 261–270 (1993)
8. Detlefs, D., Flood, C.H., Garthwaite, A., Martin, P., Shavit, N., Steele, G.L. Jr.: Even better DCAS-based concurrent deques. In: Proceedings of the 14th International Conference on Distributed Computing (DISC), pp. 59–73 (2000)
9. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Steele, G.L. Jr.: DCAS is not a silver bullet for nonblocking algorithm design. In: Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 216–224 (2004)
10. Fich, F.E., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free step complexity: Lock-free DCAS as an example. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC), pp. 493–494 (2005)
11. Greenwald, M.: Non-blocking synchronization and system design. PhD thesis, Stanford University, August 1999
12. Greenwald, M.: Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC), pp. 260–269 (2002)
13. Ha, P.H., Tsigas, P., Wattenhofer, M., Wattenhofer, R.: Efficient multi-word locking using randomization. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 249–257 (2005)
14. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of the 15th International Conference on Distributed Computing (DISC), pp. 300–314 (2001)
15. Harris, T.L., Fraser, K.: Language support for lightweight transactions. In: Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 388–402 (2003)
16. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
17. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS), pp. 522–529 (2003)
18. IBM. IBM System/370 Extended Architecture, Principle of Operation. IBM Publication No. SA22-7085 (1983)
19. Merritt, R.: IBM plants transactional memory in CPU. EETimes, August 2011. http://www.eetimes.com/electronics-news/4218914/IBM-plants-transactional-memory-in-CPU

20. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the 14th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 73–82 (2002)
21. Michael, M.M.: CAS-based lock-free algorithm for shared deques. In: Proceedings of the 9th Euro-Par Conference on Parallel Processing, pp. 651–660 (2003)
22. Reinders, J.: Transactional synchronization in Haswell, February 2012. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/
23. Schneider, J., Wattenhofer, R.: Bounds on contention management algorithms. In: Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC), pp. 441–451 (2009)
24. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
25. Sundell, H.: Efficient and practical non-blocking data structures. PhD thesis, Chalmers University of Technology (2004)
26. Sundell, H., Tsigas, P.: Lock-free deques and doubly linked lists. J. Parallel Distrib. Comput. **68**(7), 1008–1020 (2008)
27. Turek, J., Shasha, D., Prakash, S.: Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 212–222 (1992)