

# Built-in Coloring for Highly-Concurrent Doubly-Linked Lists\*

(Preliminary Version)

Hagit Attiya and Eshcar Hillel  
Department of Computer Science  
Technion

September 11, 2006

## Abstract

This paper presents a novel approach for lock-free implementations of concurrent data structures, based on dynamically maintaining a *coloring* of the data structure's items. Roughly speaking, the data structure's operations are implemented by acquiring virtual locks on several items of the data structure and then making the changes atomically; this simplifies the design and provides clean functionality. The virtual locks are managed with CAS or DCAS primitives, and helping is used to guarantee progress; virtual locks are acquired according to a coloring order that decreases the length of waiting chains and increases concurrency. Coming back full circle, the legality of the coloring is preserved by having operations correctly update the colors of the items they modify.

The benefits of the scheme are demonstrated with new nonblocking implementations of doubly-linked list data structures: A DCAS-based implementation of a doubly-linked list allowing insertions and removals anywhere, and CAS-based implementations in which removals are allowed only at the ends of the list (insertions can occur anywhere).

The implementations possess several attractive features: they do not bound the list size, they do not leave accessible chains of garbage nodes, and they allow operations to proceed concurrently, without interfering with each other, if they are applied to non-adjacent nodes in the list.

**Keywords:** local step complexity, local contention, nonblocking implementations, CAS, DCAS, doubly-linked list, double-ended queue, priority queue.

---

\*This research was supported by the *Israel Science Foundation* (grant number 953/06)

# 1 Introduction

Many core problems in asynchronous multiprocessing systems revolve around the coordination of access to shared resources and 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 items at the two ends of the queue; it can be used as a producer-consumer job queue [3]. A *priority queue* can be implemented as a doubly-linked list where removals are allowed only at the ends, while items 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, often called simply a *linked list*) allows insertions and removals anywhere in the linked list.

Concurrent data structures are implemented by applying *primitives*—provided by the hardware or the operating system—to memory locations. *Lock-free implementations* do not rely on mutual exclusion, thereby avoiding the inherent problems associated with locking—deadlock, convoying, and priority-inversion. Lock-free implementations must rely on strong primitives [14], e.g., CAS (*compare and swap*) and its multi-location variant, *kCAS*.

Lock-free implementations are often complex and hard to get right; even for relatively simple, key data structures, like deques, they suffer from significant drawbacks: Some implementations may contain garbage nodes [13], others statically limit the data structure’s size [15] or do not allow concurrent operations on both ends of the queue [19]. Even when DCAS (i.e., 2CAS) is used, existing implementations either are inherently sequential [10, 11] or allow to access chains of garbage nodes [8].

Implementing concurrent data structures is fairly simple if an arbitrary number of locations can be accessed atomically. For example, removing an item from a doubly-linked list is easy if one can atomically access three items—the item to be removed and the two items before and after it (cf. [8]).

Since no multiprocessor supports primitives that access more than two locations atomically, it is necessary to simulate them in software using CAS or DCAS. This can be done using methods such as *software transactional memory* [21] or the so-called *locking without blocking* techniques [6, 24]. The basic idea of these methods is to use CAS in order to acquire *virtual locks* on the items—one item at a time, and *help* processes that hold virtual locks on desired items until they are released. This guarantees that the simulation is *nonblocking* [14], namely, in any infinite execution, some pending operation completes within a finite number of steps. Unfortunately, the resulting implementations may have long waiting chains, creating interference among operations and reducing the implementation’s throughput.

Attiya and Dagan [4] suggest an alternative implementation of binary operations that reduces interference by using *colors* (from a small set). This *color-based virtual locking* scheme starts by legally coloring the items it is going to access, so that neighboring items have distinct colors. Then, the algorithm acquires the virtual locks in increasing order of colors, thereby avoiding long waiting chains. Afek et al. [1] extended this implementation to arbitrary *k*-ary operations.

Afek et al. [1] define two measures to evaluate whether operations that access disjoint parts of the data structure, or are widely separated in time, do not interfere with each other. These definitions rely on the notion of a *conflict graph*, whose nodes are the data items and there is an edge between two items if they are accessed by the same operation. Roughly speaking, the *distance* between operations in the conflict graph is the length of the shortest path between their data items. An implementation has *d-local step complexity* if only operations in distance less than or equal to *d* in the conflict graph can delay each other; it has *d-local contention* if only operations in distance less than or equal to *d* in the conflict graph can access the same locations simultaneously.<sup>1</sup> In particular, when there is no path in the conflict graph between the data items

---

<sup>1</sup>Attiya and Dagan [4] used a more complicated measure called *sensitivity*, which is not discussed in this extended abstract.

accessed by two operations, they do not delay each other or access the same memory locations; thus,  $d$ -local step complexity and contention extend and generalize *disjoint-access parallelism* [18].

The implementations [1, 4] have  $O(\log^* n)$ -local step complexity and contention, and they are rather complicated, making them infeasible for fundamental linked list-based data structures. The major reason for the cost and complication of these implementations is the need to color memory locations at the beginning of each operation, since operations access arbitrary and unpredictable sets of memory locations.

When operations are applied on a specific data structure, however, they access its constituent items in a predictable, well-organized manner; e.g., linked list operations access two or three consecutive items. In this case, why color the accessed items from scratch, each time an operation is invoked? After all, the implementation initializes the data structure and provides operations that are the only means for manipulating it. If the colors are built into the items, then an operation can rely on them to guide its locking order, without coloring them first. In return, the operation needs to guarantee that the modifications it applies to the data structure preserve the legality of the items' coloring.

We demonstrate this approach with two new doubly-linked list algorithms: A CAS-based implementation in which removals are allowed only at the ends of the list (and insertions can occur anywhere), and a DCAS-based implementation of a doubly-linked list allowing insertions and removals anywhere.

The CAS-based implementation, allowing insertions anywhere and removals at the ends, is based on a 3-coloring of the linked list items. It has 4-local contention and 4-local step complexity.; namely, an operation only contends with operations on items close to its own items on the linked list, and it is delayed only due to such operations. When insertions are also limited to occur at the ends, the analysis can be further refined to show 2-local contention and 2-local step complexity; this means that operations at the two ends of a deque containing three data items (or more) never interfere with each other.

Handling removals from the middle of the linked list is more difficult: removing an item might entail recoloring one of its neighbors, requiring to make sure its neighbor's color is not changed concurrently. Thus, a remove operation has to lock *three* consecutive items; under a legal coloring it is possible that two of these items (necessarily non-consecutive) have the same color. We employ a DCAS operation to lock these two nodes atomically, thereby avoiding hold-and-wait chains. This algorithm has 6-local contention and 2-local step complexity. To the best of our knowledge, this is the first nonblocking implementation of a doubly-linked list from realistic primitives, which allows insertions and removals anywhere in the list, and has low interference.

In our algorithms, an operation has constant *obstruction-free step complexity* [9]; namely, an operation completes within  $O(1)$  steps in an execution suffix in which it is running solo. Another attractive feature of our implementations is that it does not leave accessible chains of stale "garbage" nodes.

In recent years, a flurry of papers proposed implementations of dynamic linked list data structures, yet none of them provided all the features of our algorithms. (See Table 1.) Some of these algorithms have significant drawbacks, e.g., they are incompatible with garbage collectors, or inherently sequential, or bound the linked list's size.

Harris [13] used CAS to implement a singly-linked list, with insertions and removals anywhere; however, in this algorithm, a process can access a node previously removed from the linked list, possibly yielding an unbounded chain of uncollected garbage nodes. Michael [20] handled these memory management issues. Elsewhere [19], Michael proposed an implementation of a deque; in his algorithm, a single word (called *anchor*) holds the head and tail pointers, causing all operations to interfere with each other, thereby making the implementation inherently sequential. Sundell and Tsigas [23] avoid the use of a single anchor, allowing operations on the two ends to proceed concurrently. They extend the algorithm to allow insertions and removals in the middle of the list [22]; in the latter algorithm, a long path of overlapping removals may

algorithm	insertions	deletions	primitive	interference	comments
Greenwald List [10]	anywhere	anywhere	DCAS	any pair	
Harris [13]	anywhere	anywhere	CAS	any pair	singly-linked list chains of garbage nodes
Michael Set [20]	anywhere	anywhere	CAS	any pair	singly-linked list
Sundell and Tsigas [23]	anywhere	anywhere	CAS	any pair	
Alg. DCAS-CHROMO	anywhere	anywhere	DCAS	distance $\leq 2$	
Michael Deque [19]	ends	ends	CAS	opposite ends	doubly-linked list
Herlihy et al. Deque [15]	ends	ends	CAS	distance $\leq 1$	obstruction free size is bounded
Greenwald Deque [11]	ends	ends	DCAS	opposite ends	
Agesen et al. Deque [2]	ends	ends	DCAS	distance $\leq 1$	doubly-linked list
Sundell and Tsigas [23]	ends	ends	CAS	distance $\leq 1$	
Alg. CAS-CHROMO	anywhere	ends	CAS	distance $\leq 4$	
Alg. CAS-CHROMO	ends	ends	CAS	distance $\leq 2$	

Table 1: Linked list algorithms; *interference* indicates which operations may delay other operations.

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 liveness property weaker than nonblocking, was proposed by Herlihy et al. [15]; besides blocking when there is even a little contention, this array-based implementation bounds the deque’s size.

Greenwald [10, 11] suggests to use DCAS to simplify the design of implementations 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 of operations. Agesen et al. [2] present the first DCAS-based nonblocking, dynamically-sized deque implementation that supports concurrent access to both ends of the deque, and has 1-local step complexity; this algorithm does not allow insertions or removals in the middle of the linked list. The SNARK algorithm [7] 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 [8]; 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. Doherty et al. [8] even argue that primitives more powerful than DCAS, e.g., 3CAS, are needed in order to obtain simple and efficient nonblocking implementations of concurrent data structures.

The rest of this paper is organized as follows. Section 2 presents the model of a asynchronous shared-memory system, while Section 3 defines local contention and local step complexity in a dynamic setting. Most of the paper describes the DCAS-based implementation of a doubly-linked list allowing insertions and removals anywhere (Section 4). Section 6 outlines the modifications needed to obtain the CAS-based implementation that does not allow removals from the middle.

## 2 Preliminaries

We consider a standard model for a shared memory system [5] in which a finite set of *asynchronous processes*  $p_1, \dots, p_n$  communicate by applying *primitive* operations to  $m$  shared *memory locations*,  $l_1, \dots, l_m$ .

A *configuration* is a vector  $C = (q_1, \dots, q_n, v_1, \dots, v_m)$ , where  $q_i$  is the local state of  $p_i$  and  $v_j$  is the

value of memory location  $l_j$ .

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

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

A *data structure* of type  $T$  supports a set of operations that provide the only means to manipulate it. Each data structure has a *sequential specification*, which indicates how it is modified when operations are applied in a serial manner (in isolation).

An *implementation* of a data structure  $T$  provides a specific data-representation for  $T$ 's instances as a set of memory locations, and protocols that processes must follow to carry out  $T$ 's operations, defined in terms of primitives applied to memory locations. We require the implementation to be *linearizable* [16].

This paper considers a *doubly-linked list* data structure, composed of *nodes*, each with link pointers to its left and right neighboring nodes. Two special *anchor* nodes serve as the first (leftmost) and last (rightmost) nodes in the doubly-linked list; they cannot be removed from it, and hold no left link or no right link, respectively. A node is *valid* in configuration  $C$  if it is either an anchor, or both its left link and right link pointers are not null.

We concentrate on the *InsertRight*, *InsertLeft* and *Remove* operations applied to some *source* node in the linked list. Our description of their effects resembles the description of the deque operations in [2]:

**insertRight( $nd$ )** If *source* is a valid node other than the right anchor, then insert  $nd$  to the right of *source* and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

**insertLeft( $nd$ )** If *source* is a valid node other than the left anchor, then insert  $nd$  to the left of *source* and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

**remove()** If *source* is a valid node other than an anchor, then remove *source* from the linked list and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

In order to apply an operation to the data structure, process  $p_i$  executes the associated protocol. We denote an operation of process  $p_i$  by  $op_i$ ; the  $j$ -th operation of  $p_i$  is denoted  $op_i^j$ .

The *interval of an operation*  $op$ , denoted  $I_{op}$ , is the execution interval between the first and last events of the process executing  $op$ 's protocol; if the operation does not terminate, its interval is infinite. Two operations *overlap* if their intervals overlap. The *interval of a set of operations*  $OP$ , denoted  $I_{OP}$ , is the minimal execution interval that contains all intervals,  $\{I_{op}\}_{op \in OP}$ .

### 3 Locality Properties

The *reference lock-based implementation* of a data structure  $T$  atomically locks all the memory locations that it accesses; these are called the *lock set* of the operation. Different lock-based implementations may have different lock sets. Since we aim for highly concurrent implementations, we choose a reference implementation that locks as few data items as possible; for a linked list data structure this number is a constant.

When operations are applied sequentially the state of the data structure is well defined at the end of each operation. The lock set of an operation  $op_i$  applied when the data structure is in state  $s$  is denoted  $\mathcal{L}S_s(op_i)$ .

When operations are concurrent, the state of the data structure at a configuration  $C$  is not necessarily unique. A state  $s$  of the data structure is *possible* in configuration  $C$ , if it is the result of some linearization that includes all operations that complete before  $C$  and a subset of the operations that are pending in  $C$ . The set of all possible states in  $C$  is denoted  $states(C)$ .

Intuitively, the data set of an operation includes all the data items the operation accesses. When the data structure is dynamic, however, the data set changes over time and it is unknown when the operation is invoked. For this reason, we consult the reference implementation regarding the data items it locks with respect to all the states of the data structure during the operation's interval. Formally, the *data set* of an operation  $op_i$  in configuration  $C$  is defined as  $\mathcal{DS}_C(op_i) = \bigcup_{s \in states(C)} \mathcal{LS}_s(op_i)$ ; i.e., the union of all the sets of data items the operation locks (under the reference implementation) when the state of the data structure is in  $states(C)$ .  $\mathcal{DS}(op_i) = \bigcup_{C \in I_{op_i}} \mathcal{DS}_C(op_i)$ ; namely, the union of  $\mathcal{DS}_C(op_i)$  over all configurations during  $op_i$ 's execution interval.

For example, Figure 1(a) depicts several overlapping operations;  $op_1$ ,  $op_3$ , and  $op_5$  insert a new node to the right of  $m_2$ ,  $m_4$ , and  $m_8$ , respectively, while  $op_2$  and  $op_4$  remove  $m_3$  and  $m_6$  respectively. The new node, omitted from the figure, is also in the operation's data set. Consider the following execution: in the first execution interval  $C_0, \dots, C_1$  every operation takes one step, and assume for simplicity it is a read step; then, in the execution interval  $C_1, \dots, C_2$  only  $op_2$  takes steps until it completes successfully to remove  $m_3$ ; finally, in the execution interval  $C_2, \dots, C_3$  only  $op_1$  takes steps until it completes successfully. Figure 1(a) depicts the data set of the operations at configuration  $C_1$ . Note that the data set of  $op_1$  changes over time:  $\mathcal{DS}_{C_1}(op_1) = \{m_2, m_3\}$  while  $\mathcal{DS}_{C_2}(op_1) = \{m_2, m_4\}$ , and altogether the data set of the operation is  $\mathcal{DS}(op_1) = \{m_2, m_3, m_4\}$ .

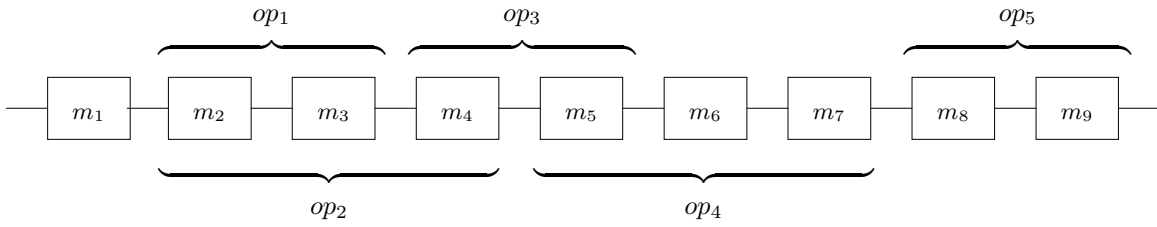
The *conflict graph* of a configuration  $C$ , denoted  $G(C)$ , is an undirected graph that captures the distance between overlapping operations. If  $C$  is in the execution interval of an operation  $op_i$ , and  $m_v$  and  $m_u$  are data items in  $\mathcal{DS}_C(op_i)$ , then the conflict graph includes an edge between the respective vertices  $v$  and  $u$ , labeled  $op_i$ . Figures 1(b) and 1(c) depict the conflict graph of configurations  $C_1$  and  $C_2$ , respectively. The conflict graph of an execution interval  $\alpha$  is the graph  $\bigcup_{C \in \alpha} G(C)$ . Figure 1(d) depicts the conflict graph of  $op_1$ 's execution interval.

The *conflict distance* (in short *distance*) between two operations,  $op_i$  and  $op_j$ , in a conflict graph is the length (in edges) of the shortest path between some vertex in  $\mathcal{DS}(op_i)$  and some (possibly the same) vertex in  $\mathcal{DS}(op_j)$ . In particular, if  $\mathcal{DS}(op_i)$  intersects  $\mathcal{DS}(op_j)$ , then the distance between  $op_i$  and  $op_j$  is zero. The distance is  $\infty$ , if there is no such path. In the conflict graph of configuration  $C_1$  (Figure 1(b)), the distance between  $op_1$  and  $op_2$  is zero, 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  $op_5$  is  $\infty$ . After  $op_2$  completes in configuration  $C_2$  (Figure 1(c)), the distance between  $op_1$  and other operations is decreased by one, since at this configuration the data set of  $op_1$  includes  $m_4$ . In the conflict graph of  $op_1$ 's execution interval (Figure 1(d)), the distance between  $op_1$  and  $op_3$  is zero, the distance between  $op_1$  and  $op_4$  is one, and the distance between  $op_1$  and  $op_5$  remains  $\infty$ .

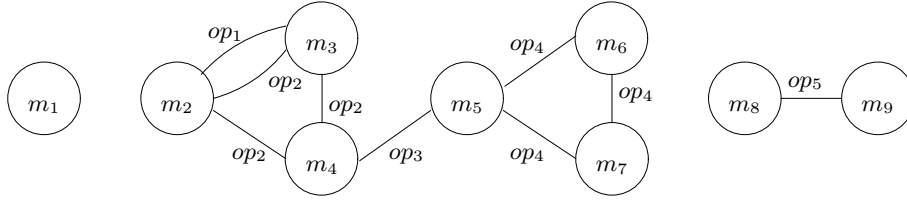
We use this dynamic version of a conflict graph in the definitions of locality measures suggested by Afek et al. [1]:

**Definition 1** *An algorithm has  $d$ -local step complexity if the number of steps performed by process  $p$  during the operation interval  $I_{op}$  is bounded by a function of the number of operations at distance smaller than or equal to  $d$  from  $op$  in the conflict graph of its operation interval  $I_{op}$ .*

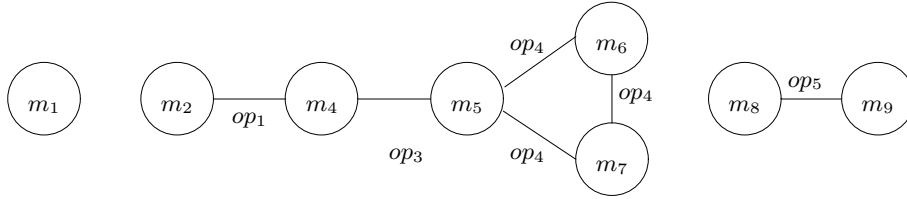
**Definition 2** *An algorithm has  $d$ -local contention if in every execution interval for any two operations,  $I_{\{op_1, op_2\}}$ ,  $op_1$  and  $op_2$  access the same memory location only if their distance in the conflict graph of  $I_{\{op_1, op_2\}}$  is smaller than or equal to  $d$ .*



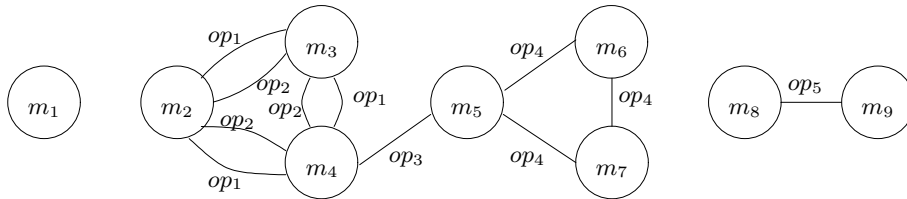
(a) Example of overlapping operations on a linked list.



(b) The conflict graph  $G(C_1)$ .



(c) The conflict graph  $G(C_2)$ .



(d) The conflict graph  $G(I_{op_1})$ .

Figure 1: Simple examples of conflict graph.

## 4 DCAS-Based Doubly-Linked List Algorithm

We demonstrate our approach with a nonblocking implementation, DCAS-CHROMO, of a doubly-linked list with insertions and removals anywhere. At the heart of our methodology is an enhancement of the colored-based virtual locking scheme. We first review this scheme, and then describe our algorithm.

### 4.1 The Color-Based Virtual Locking Scheme

Data structures can be implemented by the nonblocking *virtual locking* scheme [6, 21, 24]. A concurrent implementation is systematically derived from any lock-based algorithm: an operation starts by acquiring *virtual locks* on the data items in its data set (LOCK phase); then, the appropriate changes are applied on these data items (APPLY phase); finally, the operation releases the virtual locks (UNLOCK phase). Similar to a lock-based solution, while a data item is locked by an operation, other operations can neither lock nor modify it. This means the algorithm is relieved of handling inconsistent states due to contention.

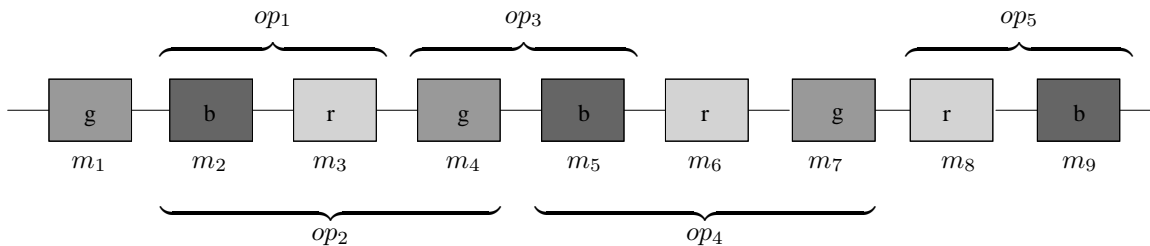


Figure 2: 3-Coloring of the linked list in Figure 1(a).

An operation is *blocked* if a data item in its data set is locked by another, *blocking* operation. In order to make the scheme nonblocking, the process executing the blocked operation  $op$  helps the blocking operation  $op'$  to complete and release its data set. Several processes may execute an operation; the process that invokes the operation is its *initiator*, while the *executing processes* are processes helping the initiator to complete or the initiator itself. CAS primitives are used to guarantee that only one of the executing processes performs each step of the operation, and others have no effect.

This scheme induces *recursive* helping, in which one process helps another process to help a third process and so on, possibly causing long helping chains. For example, assume the nodes in Figure 1(a) are locked in ascending order. Consider an execution  $\alpha$  in which  $op_2$ ,  $op_3$  and  $op_4$  concurrently lock their left-most data items successfully, and then  $op_1$  tries to lock its data items while the other operations are delayed. Since  $m_2$  is locked by  $op_2$ ,  $op_1$  has to help  $op_2$ ; since  $m_4$  is locked by  $op_3$ ,  $op_1$  has to help  $op_3$ ; and since  $m_5$  is locked by  $op_4$ ,  $op_1$  has to help  $op_4$ . Thus  $op_1$  is delayed by operations on a path in  $\alpha$ 's conflict graph, from some vertex in  $\mathcal{DS}(op_1)$ . In general,  $op_1$  can be delayed by any operation within finite distance from it, implying that the locality is high.

Shavit and Touitou [21] overcome this problem by helping only an immediate neighbor in the conflict graph. Nevertheless, the number of steps a process performs depends on the length of the longest path from its data set in the conflict graph. Consider again an execution that starts with  $op_2$ ,  $op_3$  and  $op_4$  locking their low-address data items successfully, then  $op_1$  fails to lock  $m_2$ ,  $op_2$  fails to lock  $m_4$ , and  $op_3$  fails to lock  $m_5$ ; each operation then helps its (immediate) neighbor. Prior to helping,  $op_2$  and  $op_3$  relinquish their locks and fail, 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 lock their data sets. It is possible that  $op_2$  and  $op_3$  lock their low-address data items, and  $op_1$  tries, in vain, to help  $op_2$ , which releases its locks due to  $op_3$ , etc. As the length of the path of overlapping operations increases, the number of times  $op_1$  futilely helps  $op_2$  increases as well.

A *color-based* virtual locking scheme [4] bounds the length of helping chains by  $M$ -*coloring* the data items with an ordered set of colors,  $c_1 < c_2 < \dots < c_M$ . An operation acquires locks on data items in an increasing order of colors; after it locks all  $c_i$ -colored data items, we say the operation *locked color*  $c_i$ . In this scheme,  $op$  helps  $op'$  only if  $op'$  already locked a higher color.

Figure 2 presents a 3-coloring of the linked list in Figure 1(a) using the colors  $r(\text{red}) < g(\text{green}) < b(\text{blue})$ . Assume  $op_3$  locks  $m_4$  and then tries to lock  $m_5$ , with color  $b$ . If the lock on  $m_5$  is already held by  $op_4$ , then  $op_3$  has to help  $op_4$ . Note however, that  $b$  is the largest color, which means that  $op_4$  already locked all the nodes in its data set. This means that  $op_3$  will only have to apply  $op_4$ 's changes, and  $op_3$  is not required to recursively help additional operations. Along these lines, it is possible to prove that the length of helping chains is bounded by the number of colors,  $M$ , and the number of times an operation helps other operations is bounded by a function of the number of operations within distance  $M$  [4].

Originally [1, 4], colors were assigned to nodes from scratch each time an operation starts. This is done in a DECISION phase, which obtains information about operations (and their data sets) at non-constant distance; thus, the DECISION phase has non-constant locality properties.

## 4.2 Our Approach

We achieve constant locality properties by employing two complementary algorithmic ideas. The first is to maintain the data structure legally colored at all times, and the second is to atomically lock all data items with the same color.

The key idea of our approach is to keep the coloring legal while the operation is in its APPLY phase, rendering the DECISION phase obsolete. That is, the colors are built into the nodes, and the operation updates the colors so that nodes remain legally colored. These changes are limited to the nodes in the operation's data set, and bypass the need to re-compute a legal coloring from scratch each time an operation is invoked.

The second idea avoids long helping chains due to symmetric color assignments. For example, consider a long legally colored linked list of nodes with alternating colors:  $b, r, b, r, b, r, \dots$ . Assume a set of concurrent operations, each of which is trying to remove a different  $r$ -colored node, by first locking the node and its two  $b$ -colored neighbors. An implementation that locks these two  $b$ -colored nodes one at a time, e.g., first the left neighbor, can lead to a configuration in which an operation holds its left lock, and needs to help all operations to its right.

It is tempting to extend the notion of a legal coloring and require that any triple of neighboring nodes is assigned distinct colors. This certainly will allow to follow the color-based virtual locking scheme, but how can we preserve this extended coloring property? In particular, when a node is removed, it is necessary to lock *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 lock *five* nodes and so on.

Locking equally-colored nodes atomically provides an escape from this vicious circle, by avoiding this situation altogether. An operation accesses at most three consecutive nodes, which are legally colored, thus at most two of these nodes have the same color, and a DCAS suffices for locking them. For example, in the scenario described above, locking the two  $b$ -colored nodes atomically breaks the symmetry. This guarantees that the LOCK phase has  $O(1)$ -local step complexity.

Another aspect of our algorithm is in handling the complications due to dynamically-changing data structures. Previous implementations of the virtual locking scheme handle static transactions [21] and multi-location operations [1, 4]; in both cases, an operation accesses a pre-determined static data set.

Our algorithm addresses this problem in a manner similar to [12]. A *data set memento*, which holds a view of the data set when the operation starts, traces inconsistencies in the data set due to changes applied by concurrent operations. If, while locking, the operation detects such inconsistency, the operation skips the APPLY phase to the UNLOCK phase where it releases all the locks it holds. If, on the other hand, the operation completes its LOCK phase, then the locked data set memento is consistent with the operation data set and the operation can continue with the APPLY phase as in a static virtual locking scheme.

## 4.3 Detailed Description of Algorithm DCAS-CHROMO

First, we describe how operations apply their changes to the data structure, and give some intuition of how the legal coloring is preserved; then we describe the helping mechanism that is responsible for the nonblocking and locality properties.

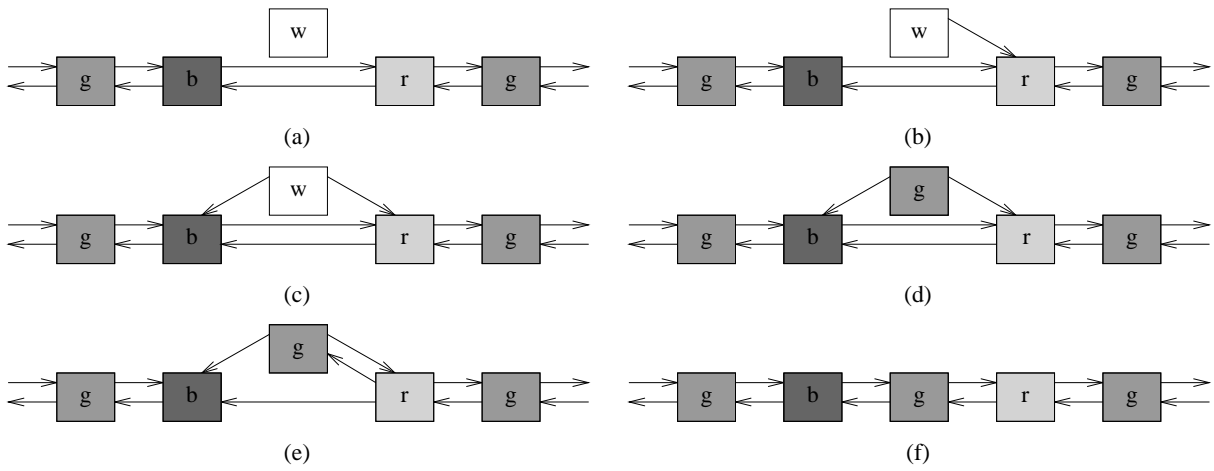


Figure 3: An example of an *InsertRight* operation -  $op_1$  in Figure 2

The lock-based implementation we use as a reference has the following lock sets: An *InsertRight* operation locks the new node to be inserted, the *source* node (to which the operation is applied) and its right neighbor; an *InsertLeft* operation is symmetric. A *Remove* operation locks the *source* node and both its left and right neighbors. After locking, the operations apply changes to the respective set of left and right links as described by the following code:

```

InsertRight::applyChanges() {
    newNode.right ← source.right
    newNode.left ← source
    source.right.left ← newNode
    source.right ← newNode
}

```

```

Remove::applyChanges() {
    source.left.right ← source.right
    source.right.left ← source.left
    source.right ← ⊥
    source.left ← ⊥
}

```

Since our algorithm employs a virtual locking scheme, each operation proceeds in exclusion in a manner similar to the lock-based one. Our implementation, however, also needs to maintain the nodes legally colored. This requires adding one step to the *InsertRight* operation (see Figure 3), and two steps to the *Remove* operation (see Figure 4). To ensure that the coloring is legal at all times, we use a temporary color  $c_0 < c_1$  during the algorithm, as described bellow. In the example figures,  $c_0$  is  $w$  (*white*).

***InsertRight* operation.** Figure 3(a) presents the nodes  $m_1, m_2, m_3, m_4$  from Figure 2, and the new node,  $m$ , that  $op_1$  inserts to the right of  $m_2$ . Before  $m$  is inserted to the linked list, it is colored with the temporary color,  $w$ .  $op_1$  locks the nodes in its data set,  $m_2$  and  $m_3$  (and effectively, also  $m$ ), and then applies its changes as follows: update right and left links of  $m$  to point to  $m_3$  and  $m_2$ —now,  $m$  is legally colored, since its neighbors have colors different than  $w$  (Figures 3(b),3(c));  $m$  is assigned with a non-temporary color different than its neighbors  $m_2$  and  $m_3$  (Figure 3(d)); update left and right links of  $m_3$  and  $m_2$  respectively to point to  $m$  (Figures 3(e),3(f)).

***Remove* operation.** Figure 4(a) presents the nodes  $m_1, m_2, m_3, m_4, m_5$  from Figure 2,  $op_2$  removes the node  $m_3$ .  $op_2$  locks the nodes in its data set,  $m_2, m_3$  and  $m_4$ , before it applies its changes as follows:  $m_4$  is assigned with the temporary color,  $w$ —now,  $m_4$  is legally colored, since its neighbors  $m_3$  and  $m_5$  have colors different than  $w$  (Figure 4(b)); update right and left links of  $m_2$  and  $m_4$  respectively to point to each other (Figures 4(c),4(d)); set right and left links of  $m_3$  to null (Figure 4(e));  $m_4$  is assigned with a non-temporary color different than its neighbors  $m_2$  and  $m_5$  so it is legally colored (Figure 4(f)).

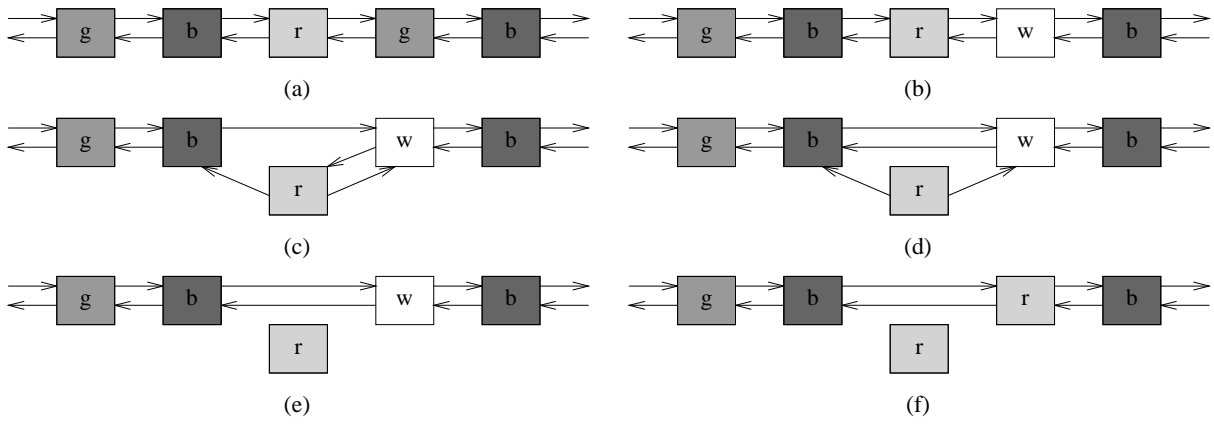


Figure 4: An example of a *remove* operation -  $op_2$  in Figure 2

Both an *InsertRight* operation and a *Remove* operation access three consecutive nodes in the data set, however, each operation only changes the color of a single node. An *InsertRight* operation changes the color of the new node, and a *Remove* operation changes the color of the right node. The color of the left node in the data set of any operation is not modified. This ensures that no two adjacent nodes change their color concurrently even if they belong to the data sets of two adjacent concurrent operations.

We now detail the color-based locking and helping mechanisms. An operation is partitioned into *invocations*, each invocation has a unique sequence number. Figure 5 shows the state transition diagram of an operation's invocation. The dashed line indicates re-invocation, increasing the sequence number. The state transitions of an invocation in a best-case execution, encountering no contention, appear at the top. In such case the invocation *completes successfully* and the operation will not be re-invoked. If an operation discovers, while initiating an invocation, that another operation removed the source node, meaning the source is no longer a valid list node, then it need not apply its changes, and it will not be re-invoked. Alternatively, if, while locking the data set, an operation discovers that a node in its data set memento is invalid or if the operation detects inconsistency with the data set memento, then the operation needs to re-evaluate its data set. In such a case, the invocation *fails*, the operation releases the locks it already acquired and restarts a new invocation.

The state of an operation is a tuple  $\langle seq, phase, result \rangle$  (see Algorithm 1): *seq* is an integer, initially 0, incremented every time the operation fails and the initiator process reinvokes it; *phase* indicates the locking scheme phase within the invocation, set to INIT at the beginning of every invocation; *result* holds the result of the current invocation execution, set to NULL at the beginning of every invocation.

Algorithm 2 lists the locking methods. Their core is the `lockColor` method in which an operation  $op$  repeatedly tries to atomically lock all nodes with color  $c$  in its current data set memento (lines lc4-lc15); after verifying the mementos are valid and the nodes are consistent with their mementos (line lc6),  $op$  tries to lock them atomically (line lc9). If when an operation  $op$  fails to lock the nodes it discovers that none of them is locked by another operation, it simply retries to acquire their locks. Alternatively,  $op$  may discover that a node in its data set is locked by another, blocking operation  $op'$  (line hb2). In such a case, we follow the standard recursive helping mechanism, i.e.,  $op$  helps  $op'$  (line hb4). Before helping  $op'$ , the executing process of  $op$  verifies (again) that the nodes are consistent with their mementos (line lc12). This is crucial for maintaining the locality properties of the algorithm (see Lemma 10 in the next section).

If, when initiating an invocation, an operation discovers that its source node is invalid (line t2), it helps the operation that removes this node (line t4), in order to preserve the correct order in which the operations

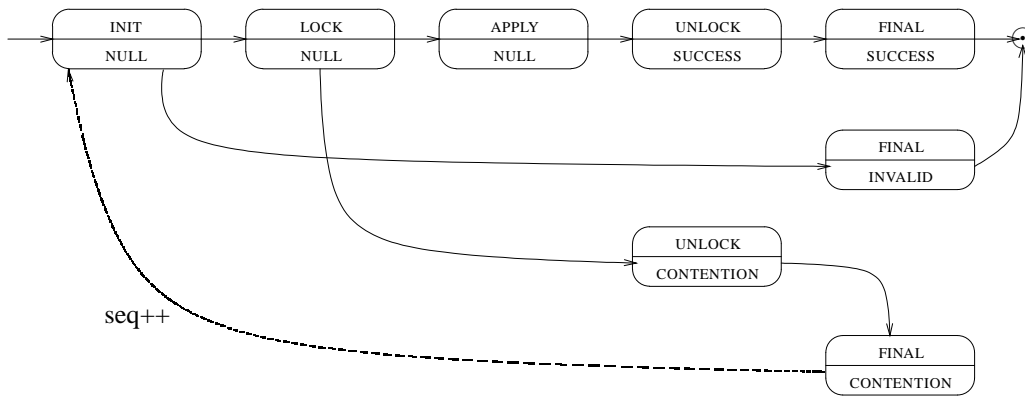


Figure 5: Diagram of an operation state transitions model; the lower part of the state is the value of *result*.

complete (see Lemma 3 in the next section).

Since an operation may be invoked more than once, its execution is composed of an alternating sequence of acquiring and releasing locks. Hence, having more than one process execute the operation requires special care. Specifically, a process may acquire locks of previous invocations or release locks acquired in a later invocation. Together with the CAS primitives, the state is used to synchronize between the executing processes of an operation. An executing process helps to execute some invocation of an operation (line t9). Before acquiring a lock on behalf of the operation, the process verifies that the current sequence number is equal to the invocation it is executing (line lc8). Furthermore, to prevent a process from releasing locks acquired in a later invocation, the operation stamps any lock it acquires with its sequence number (line ln9 or ln13). Before a process releases a lock, it verifies that the sequence number stamped on the lock is equal to the invocation it is executing (line ul8).

#### 4.4 Implementation Details

We use object-oriented terminology and define operations as objects, whose structure and behavior are defined in the *Operation* hierarchy.

A process initializes an operation object with all the data required for its execution, specifically the source node from the linked list on which the operation is applied. Algorithm 1 presents the generic protocol for an operation execution. The execution starts with the `execute` method (line ex1) and as long as it suffers from contention and is unable to complete, the process repeatedly tries to clear the operation’s attributes and re-invoke it (lines ex3-ex5): First it generates the new data set memento (line t6); then it “helps” itself to follow the locking scheme (line t9); locks nodes in its data set (line h2), applies its changes (line h5), and releases the data set (line h7). Specific operations, such as *InsertRight* and *Remove*, extend the *Operation* structure and refine its protocols for cloning and manipulating the data set with respect to their specifications (Algorithm 3).

Algorithm 1 also lists the main data structures, representing nodes and operations. Each node has a *data* attribute, containing its information. A node also contains two link references *left* and *right*, with the obvious meaning, as well as *color* and *lock* attributes. A node memento contains a reference to the node itself, *owner*, and a copy of the node’s attributes except for the data and the lock.

It is well-known that CAS primitives suffer from the ABA problem [17]: 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

---

**Algorithm 1** Algorithm DCAS-CHROMO: Definitions and execution scheme

---

```
// Types and structures and classes definition
define Oid = int // operation id (operation address)
define Phase = {INIT, LOCK, APPLY, UNLOCK, FINAL}
define Result = {NULL, SUCCESS, CONTENTION, INVALID, EMPTY}
structure State {seq : int, phase : Phase, result : Result} // seq - for repeated invocations
structure NodeABA {node : Node, aba : int}
structure ColorABA {color : Color, aba : int}
structure LockABA {lock : Oid, seq : int, aba : int} // seq - operation's invocation number

structure Node {
  data : Data
  left : NodeABA
  right : NodeABA
  color : ColorABA
  lock : LockABA
}

structure NodeMemento {
  owner : Node
  left : NodeABA
  right : NodeABA
  color : ColorABA
}

class Operation {
  oid : Oid
  state : State // initially ⟨0,INIT,NULL⟩
  source : Node
  datasetSize : int // for a doubly-linked list, this is 3
  datasetMemento : NodeMemento[datasetSize]
  colorSet : Color[datasetSize]
}

ex1: Result Operation::execute() {
ex2:   do
ex3:     clear data set memento and color set
ex4:     toInitState()
ex5:     try()
ex6:     while state.result = CONTENTION
ex7:     return state.result
ex8: }

t1: Operation::try() {
t2:   if source is invalid then
t3:     lock ← GetLock(source)
t4:     helpBlocking(lock)
t5:     toFinalInvalidState()
t6:     cloneDataset()
t7:     seq ← state.seq
t8:     toLockState()
t9:     help(seq)
t10:    toFinalState()
t11: }

h1: Operation::help(int seq) {
h2:  lockDataset(seq) // by ascending colors
h3:  s ← state
h4:  if s.phase = APPLY then
h5:    applyChanges()
h6:    toUnlockState()
h7:    unlockDataset(seq)
h8: }

hb1: Operation::helpBlocking(Lock lock) {
hb2:  if lock != ⊥ then // locked by blocking operation
hb3:    op, opseq ← get blocking info from lock
hb4:    op.help(opseq)
hb5: }
```

---

CAS on  $l$  and the comparison succeeds whereas it should have failed. We use the simplest way to avoid this problem: associate each attribute with a monotonically increasing counter. The attribute and the counter fit into a single memory location and are manipulated atomically; the counter is incremented whenever the

---

**Algorithm 2** Algorithm DCAS-CHROMO: Locking methods

---

```
l1: Operation::lockDataset(int seq) {
l2:   if state != ⟨seq,LOCK,NULL⟩ then return
l3:   for each color  $c$  not yet locked by the operation, by ascending order do
l4:     lockColor( $c$ ,seq)
l5:   toApplyState(seq)
l6: }

lc1: Operation::lockColor(Color  $c$ , int seq) {
lc2:    $nm1, nm2 \leftarrow$  get  $seq$  node mementos with color  $c$ 
lc3:    $nd1, nd2 \leftarrow$   $nm1.owner, nm2.owner$ 
lc4:   while true do
lc5:      $lock1, lock2 \leftarrow$   $nd1.lock, nd2.lock$ 
lc6:     if  $nm1, nm2$  are invalid or  $nd1, nd2$  are not consistent with  $nm1, nm2$  then
lc7:       toUnlockContentionState(seq)
lc8:     if state != ⟨seq,LOCK,NULL⟩ then return
lc9:     lockNodes( $nd1, nd2, lock1, lock2, seq$ ) //  $nd1$  is to the left of  $nd2$ 
lc10:     $lock \leftarrow$  GetLock( $nd1, nd2$ ) // Check whether succeeded or blocked
lc11:    if  $lock.oid = oid$  then return
lc12:    if  $nd1, nd2$  are not consistent with  $nm1, nm2$  then
lc13:      toUnlockContentionState(seq)
lc14:    if state != ⟨seq,LOCK,NULL⟩ then return
lc15:    helpBlocking(lock)
lc16: }

ln1: Operation::lockNodes(Node  $nd1, Node nd2, LockABA lock1, LockABA lock2, int seq$ ) {
ln2:   if  $nd2 = \perp$  then
ln3:     lockOneNode( $nd1, lock1, seq$ )
ln4:   else
ln5:     lockTwoNodes( $nd1, nd2, lock1, lock2, seq$ )
ln6: }

ln7: Operation::lockOneNode(Node  $nd, LockABA lock, int seq$ ) {
ln8:   if  $lock = \langle \perp, \perp, t \rangle$  then
ln9:     CAS( $nd.lock, lock, \langle oid, seq, t+1 \rangle$ )
ln10: }

ln11: Operation::lockTwoNodes(Node  $nd1, Node nd2, LockABA lock1, LockABA lock2, int seq$ ) {
ln12:   if  $lock1 = \langle \perp, \perp, t1 \rangle$  and  $lock2 = \langle \perp, \perp, t2 \rangle$  then
ln13:     DCAS( $nd1.lock, nd2.lock, lock1, lock2, \langle oid, seq, t1+1 \rangle, \langle oid, seq, t2+1 \rangle$ )
ln14: }
```

---

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. Other methods prevent the ABA problem without the use of a per-attribute counter, and may be applied also to our algorithm.

It is assumed that an automatic garbage collection reclaims unreferenced objects such as nodes and operation objects. Long chains of garbage and garbage cycles are not formed since the links of removed nodes are nullified. The ABA prevention counter allows a removed node to be inserted into a linked list immediately (after setting its color to  $c_0$ ) without harming the correctness of the algorithm. However, this would violate the local contention property of the algorithm. Instead it is assumed that once a node is removed from one linked list it is not reused until reclaimed by the garbage collector.

---

**Algorithm 3** Algorithm DCAS-CHROMO: Clone, Apply and Unlocking methods

---

```
class InsertRight : Operation {
    newNode : Node
}

ci1: InsertRight::cloneDataset() {
ci2:   cloneNode(source, 0)
ci3:   cloneNode(newNode, 1)
ci4:   right ← datasetMemento[0].right.node
ci5:   cloneNode(right, 2)
ci6: }
```

```
class Remove : Operation {
    // no extension is required
}

cr1: Remove::cloneDataset() {
cr2:   cloneNode(source, 1)
cr3:   left ← datasetMemento[1].left.node
cr4:   cloneNode(left, 0)
cr5:   right ← datasetMemento[1].right.node
cr6:   cloneNode(right, 2)
cr7: }
```

```
c1: Operation::cloneNode(Node nd, int i) {
c2:   if nd = ⊥ then return
    // Assume atomic assignment
c3:   datasetMemento[i] ← ⟨nd,nd.left,nd.right,nd.data,nd.color⟩
c4:   add nd's memento color to the color collection
c5: }
```

```
ai1: InsertRight::applyChanges() {
ai2:   updateRight(1,2)
ai3:   updateLeft(1,0) // newNode is valid
ai4:   updateColor(1)
ai5:   updateLeft(2,1)
ai6:   updateRight(0,1)
ai7: }
```

```
ar1: Remove::applyChanges() {
ar2:   setTempColor(2)
ar3:   updateRight(0,2)
ar4:   updateLeft(2,0)
ar5:   updateRight(1,⊥) // source is invalid
ar6:   updateLeft(1,⊥)
ar7:   updateColor(2)
ar8: }
```

```
ur1: Operation::updateRight(int i, int j) {
    // ⊥ node memento contains ⊥ attributes
ur2:   nmi, nmj ← get i-th, j-th node mementos
ur3:   nd ← nmi.owner
ur4:   newr ← nmj.owner
ur5:   r ← nmi.right
ur6:   CAS(nd.right, r, ⟨newr,r.aba+1⟩)
ur7: }
```

```
uc1: Operation::updateColor(int i) {
uc2:   nm ← get i-th node memento
uc3:   nd ← nm.owner
uc4:   lc ← nd.left.node.color
uc5:   rc ← nd.right.node.color
uc6:   newc ← ChooseColor(lc,rc)
uc7:   c ← nm.color
uc8:   CAS(nd.color, c, ⟨newc,c.aba+1⟩)
uc9: }
```

```
sc1: Operation::setTempColor(int i) {
sc2:   nm ← get i-th node memento
sc3:   nd ← nm.owner
sc4:   c ← nm.color
sc5:   CAS(nd.color, c, ⟨c0,c.aba+1⟩)
sc6: }
```

```
ul1: Operation::unlockDataset(int seq) {
ul2:   for each node nd in seq memento do
ul3:     unlockNode(nd,seq)
ul4: }
```

```
ul5: Operation::unlockNode(Node nd, int seq) {
ul6:   lock ← nd.lock
ul7:   s ← state
ul8:   if lock = ⟨oid,seq,t⟩ and s.phase = UNLOCK then
ul9:     CAS(nd.lock, lock, ⟨⊥,⊥, t+1⟩)
ul10: }
```

---

---

**Algorithm 4** Algorithm DCAS-CHROMO: State transition methods

---

```
s1: Operation::toInitState() {
s2:   s ← state
s3:   CAS(state, s, ⟨s.seq+1,INIT,NULL⟩)
s4: }

s10: Operation::toLockState() {
s11:   s ← state
s12:   if s.phase != INIT then return
s13:   CAS(state, s, ⟨s.seq,LOCK,NULL⟩)
s14: }

s20: Operation::toApplyState(int seq) {
s21:   s ← state
s22:   if s != ⟨seq,LOCK,NULL⟩ then return
s23:   CAS(state, s, ⟨seq,APPLY,SUCCESS⟩)
s24: }

s30: Operation::toFinalState() {
s31:   s ← state
s32:   if s.phase != UNLOCK then return
s33:   CAS(state, s, ⟨s.seq,FINAL,s.result⟩)
s34: }

s5: Operation::toFinalInvalidState() {
s6:   s ← state
s7:   if s.phase != INIT then return
s8:   CAS(state, s, ⟨s.seq,FINAL,INVALID⟩) // LP1
s9: }

s15: Operation::toUnlockContentionState(int seq) {
s16:   s ← state
s17:   if s != ⟨seq,LOCK,NULL⟩ then return
s18:   CAS(state, s, ⟨seq,UNLOCK,CONTENTION⟩)
s19: }

s25: Operation::toUnlockState() {
s26:   s ← state
s27:   if s.phase != APPLY then return
s28:   CAS(state, s, ⟨s.seq,UNLOCK,s.result⟩) // LP2
s29: }
```

---

## 5 Correctness Proof

We prove that Algorithm DCAS-CHROMO is a nonblocking implementation of a doubly-linked list (Lemmas 6 and 9), allowing insertions and removals anywhere, with 2-local step complexity (Lemma 11) and 6-local contention complexity (Lemma 12).

### 5.1 Safety

The safety properties of the implementation, and in particular, its linearizability, hinge on showing that the implementation follows the virtual locking scheme. Namely, the executing processes preserve the correct transition of the operation between phases—locking, changing and releasing nodes in accordance with the operations’ phases. Most importantly, items in the data set are changed only while all of them are locked. As mentioned before, this is somewhat more complicated than in previous work [1, 4, 6, 21, 24], since the data set is dynamic.

It can be inferred directly from the state transition methods that appear in Algorithm 4, and the fact that they are the only way to make state transition, that an operation follows the state transition diagram in Figure 5. Moreover, prior to each invocation the sequence number, which is a component of the operation’s state, is increased. Hence, no operation makes a transition to the same state more than once.

The LOCK phase of the  $r$ -th invocation is called the  $r$ -th LOCK *phase*, and similarly for the other phases. The code implies that an operation is in APPLY phase at most once (since all feasible transitions from it are to a final state), in which case it completes successfully and will not be re-invoked again. We call this unique invocation the *last invocation*. The initiating process generates the data set memento once per invocation (line t6); the data set memento written in the  $r$ -th invocation is called the  $r$ -th *data set memento*; the data set

memento of the last invocation is called the *last data set memento*.

Several executing processes can make the transitions to APPLY and UNLOCK phases concurrently, but other transitions are only made by the initiating process. A transition to the APPLY phase only occurs once, in the last invocation; the method for issuing a transition to UNLOCK phase takes as argument the sequence number of the invocation, to ensure the transition occurs only if it is of the correct invocation.

In a manner that reflects the alleged circularity of our scheme, we prove four lemmas as if there is access to arbitrary  $k$ CAS primitive that can atomically lock all nodes with the same color, regardless of their number. These lemmas are used to prove that the coloring is always legal (Lemma 4), which implies that there are at most two nodes with the same color in the operation data set, and DCAS suffices.

The next two lemmas state the main invariants of the algorithm, indicating that the locking scheme is followed; they are proved by induction on the execution order.

**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,  $p_i$ , of operation  $op$  that executes `applyChanges`, first verifies that the operation is in the APPLY phase (line h4). In such case this is the last invocation 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 changes on the same nodes (from the last data set memento) and in the same order.

A process  $p$  that executes `applyChanges` first verifies the attribute to be changed is consistent with its memento, and then it applies CAS primitive to it. Since at least the ABA-prevention counter has changed, either by  $p$ 's successful CAS or another successful CAS, all the attributes to be changed by `applyChanges` are inconsistent with their mementos once  $p$  completes to execute the method.

Let  $p_j$  be the process that shifts  $op$  from APPLY phase to UNLOCK phase (line h6). Before changing the state,  $p_j$  executes `applyChanges` (line h5). If  $p_i$  applies CAS to node  $nd$  while executing `applyChanges` after the transition to UNLOCK phase occurs,  $p_i$ 's CAS fails since when the transition occurs  $nd$ 's attribute is inconsistent with its memento. ■

**Lemma 2** *Given an operation  $op$  the following claims are satisfied:*

1.  *$op$  locks nodes only when it is in LOCK phase, and during its  $r$ -th LOCK phase it only locks nodes that are in its  $r$ -th data set memento.*
2. *When an executing process returns from the method `lockColor(c,r)` either all nodes with color  $c$  in the  $r$ -th data set memento are locked by  $op$ , or  $op$  is not in the  $r$ -th LOCK phase.*
3. *If the  $r$ -th memento of a node  $nd$  in  $op$ 's data set is invalid or  $nd$  is changed after it is cloned by  $op$  in the  $r$ -th invocation, then  $op$  does not lock  $nd$  in the  $r$ -th invocation.*
4.  *$op$  shifts from the  $r$ -th LOCK 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 locked by  $op$ .*
5.  *$op$  only applies changes to nodes that are locked by it.*
6.  *$op$  releases locks only when it is in UNLOCK phase, and during its  $r$ -th UNLOCK phase it only releases locks from nodes it has locked in the  $r$ -th invocation.*

**Proof:** We prove all claims simultaneously by induction on the execution order. The base case is an empty execution; in this case all the claims are vacuously satisfied. Next we prove the induction step for each claim:

1. A process  $p_i$  executes the  $r$ -th invocation of  $op$ . Assume by contradiction it successfully locks  $nd$  although  $op$  is not in the  $r$ -th LOCK phase. First  $p_i$  read  $nd$  from  $op$ 's  $r$ -th data set memento (lines lc2-lc3). Before executing `lockNodes` (line lc9),  $p_i$  verifies that  $nd$ 's memento is valid (line lc6), and that  $op$  is in the  $r$ -th LOCK phase (line lc8). Let  $p_j$  be the executing process that shifts  $op$  from the  $r$ -th LOCK phase either to the  $r$ -th APPLY phase or to the  $r$ -th UNLOCK phase. The transition occurs after  $p_i$  verifies the phase (line lc8), and specifically after it reads  $nd$ 's lock (line lc5), but before  $p_i$  applies the locking CAS (line ln9 or ln13). If  $p_j$  shifts  $op$  to the  $r$ -th APPLY phase, by (4) all nodes in the  $r$ -th data set memento, including  $nd$ , are locked by  $op$  when  $p_j$  makes the transition. This contradicts the assumption that  $p_i$  successfully applied CAS on  $nd$ 's lock, since at least the ABA-prevention counter has changed.

Otherwise,  $p_j$  shifts to the  $r$ -th UNLOCK phase since it discovers that some node  $nd'$  in the  $r$ -th data set memento is inconsistent with its memento (lines lc6-lc7 or lines lc12-lc13). There are three cases depending on the order between the color of the nodes:

- (i)  $nd'$  and  $nd$  have the same color, and by our assumption,  $p_i$  locks them atomically (using the appropriate  $k$ CAS). By (3), the fact that  $nd'$  changed while  $op$  is in the  $r$ -th LOCK phase implies that  $p_i$  cannot successfully lock  $nd$  and  $nd'$  in this invocation, a contradiction to the assumption that  $p_i$  successfully locks  $nd$ .
  - (ii)  $nd'$  has lower color than  $nd$ . Thus,  $p_i$  executes `lockColor( $c', r$ )`, where  $c'$  is the color of  $nd'$ , before trying to lock  $nd$ . By (2),  $nd'$  was locked by  $op$  and by (3), it has not changed while  $op$  is in the  $r$ -th LOCK phase, which contradicts the assumption that  $p_j$  discovers it is inconsistent with its memento.
  - (iii)  $nd'$  has higher color than  $nd$ . Thus,  $p_j$  executes `lockColor( $c, r$ )`, where  $c$  is the color of  $nd$ , before discovering the change in  $nd'$ . By (2),  $nd$  was locked by  $op$  and, by (6), it was not released while  $op$  is in the  $r$ -th LOCK phase, which contradicts the assumption that  $p_i$  successfully locks  $nd$ .
2. An executing process of  $op$  that executes `lockColor( $c, r$ )` first verifies that  $op$  is in the  $r$ -th LOCK phase (line l2). It returns from the method in one of three cases: Two cases are when it recognizes a change in the state (line lc8 or line lc14), and it is evident by the state diagram that when returning from the method,  $op$  is no longer in the  $r$ -th LOCK phase. The third case is after verifying all the nodes with color  $c$  in the  $r$ -th data set memento are locked by  $op$  (lines lc10-lc11). Now, if when returning from the method some of the nodes are not locked by  $op$ , then, by (6), they are released by the operation that locked them, while it is in UNLOCK phase, so this case also satisfies the condition.
  3. If the  $r$ -th memento of a node  $nd$  in  $op$ 's data set is invalid no executing process tries to lock  $nd$  (lines lc6-lc7). Now, assume by contradiction that  $p_i$ , an executing process of  $op$ , successfully locks  $nd$  although it has changed after  $op$  cloned it in the  $r$ -th invocation. The change occurs after  $p_i$  verifies  $nd$  is consistent with its memento (line lc6), otherwise  $p_i$  does not try to lock  $nd$ . By (1), the change occurs while  $op$  is in the  $r$ -th LOCK phase. By Lemma 1, the change is applied by another operation  $op'$  in APPLY phase. By (5)  $nd$  was locked by  $op'$  while applying the change.  $op'$  could not have locked  $nd$  after it was locked by  $op$ , since by (6)  $op$  does not release its lock while in LOCK phase. Now, if  $op'$  locked  $nd$  before  $p_i$  read the content of  $nd$ 's lock, then  $p_i$  does not try to lock  $nd$  and does not apply the CAS. Otherwise,  $op'$  locked  $nd$  after  $p_i$  read its lock, and  $p_i$ 's CAS fails, since at least the lock ABA-prevention counter has changed. Both cases contradict the assumption that  $p_i$  successfully applies the CAS and locks  $nd$ .

4. An executing process shifts  $op$  from the  $r$ -th LOCK phase to the  $r$ -th APPLY phase only after it invokes  $\text{lockColor}(c,r)$  with all colors from the  $r$ -th color set while  $op$  is in the  $r$ -th LOCK phase. By (2), when returning from each such invocation, all relevant nodes are locked by  $op$ , and by (6), they were not released since. As the set of colors covers all nodes in the  $r$ -th data set memento, they are all locked by  $op$  while the transition occurs. Now, by (3), no node in the  $r$ -th data set memento is changed before  $op$  shifts to APPLY phase, or else  $op$  would have failed locking it. So, when the transition occurs all nodes in the  $r$ -th data set memento are locked by  $op$ , and their content is consistent with their mementos.
5. By Lemma 1,  $op$  applies changes while it is in APPLY phase and only to nodes that are in the last data set memento. By (4), when the transition to APPLY phase occurs all nodes in the  $r$ -th (last) data set memento are locked by it. Finally by (6),  $op$  does not release the nodes while it is in the APPLY phase, which means that  $op$  changes a node only when it holds its lock.
6. A process  $p_i$  executes the  $r$ -th invocation of  $op$ . If  $p_i$  reads the lock of a node  $nd$  (line ul6), and discovers it is not locked by  $op$ , then it does not try to release it. Otherwise,  $p_i$  tries to release  $nd$  (line ul9) after it verifies  $op$  is in UNLOCK phase and the node is locked by  $op$  in its  $r$ -th invocation (line ul8). Before the initiating process,  $p$ , shifts  $op$  from the  $r$ -th UNLOCK phase to the  $r$ -th FINAL phase (line t10), it invokes  $\text{unlockDataset}$  method (line h7), in which  $p$  tries to unlock all nodes in the  $r$ -th data set memento (lines ul2-ul3). All processes executing the  $r$ -th invocation try to release the same nodes from the  $r$ -th data set memento since by (1),  $op$  only locks nodes from the  $r$ -th data set memento in the  $r$ -th invocation. When  $p$  tries to release  $nd$  it either finds that  $nd$  is already not locked by  $op$ , or  $nd$  is still locked by  $op$ . In the latter case,  $nd$  is released after  $p$  applies its CAS (either by  $p$ 's successful CAS or another successful CAS). In both cases at least the lock ABA-prevention counter has changed, thus  $p_i$  cannot release  $nd$  successfully. ■

The proof that the algorithm is linearizable follows directly from these properties.

**Lemma 3 (Linearizability)** *Algorithm DCAS-CHROMO is linearizable.*

**Proof:** We linearize an operation  $op_i$  either when the transition to the state  $\langle \text{last}, \text{FINAL}, \text{INVALID} \rangle$  occurs (line s8), or when the transition to the state  $\langle \text{last}, \text{UNLOCK}, \text{SUCCESS} \rangle$  occurs (line s28). Since only one of these occurs during the execution of an operation the linearization points are well defined.

In the first case,  $op_i$  discovers that the *source* node is invalid, thus another operation  $op_j$  removes it. Before the transition to the FINAL phase occurs  $op_i$  helps  $op_j$ . Lemma 1 implies that  $op_j$  already reached its APPLY phase and  $op_i$  helps it to complete successfully in case it has not completed yet. Thus,  $op_i$  need not apply its changes, and it appears after  $op_j$  in the linearization.

In the second case, Lemma 2(3) and Lemma 2(4) imply that when the transition to the last APPLY phase occurs in configuration  $C$ , all the nodes in  $op_i$ 's last data set memento are valid, have not changed since they were cloned, and are locked by  $op_i$ . By inspection of the  $\text{cloneDataset}$  method it is easy to verify that in such case the nodes in the memento, i.e., the nodes  $op_i$  locked are the data set  $\mathcal{DS}_C(op_i)$ . By Lemma 2(6), the last data set remains locked while  $op_i$  is in APPLY phase, which means, by Lemma 2(5), that no other operation applies changes to these nodes and the data set of  $op_i$  is  $\mathcal{DS}_C(op_i)$  during the execution of the APPLY phase. Inspection of the  $\text{applyChanges}$  method shows that the changes applied by  $op_i$  preserve the sequential specification of the corresponding doubly-linked list operations. ■

Hereafter we regard the last data set memento, on which the changes are applied, as the operation's data set. The linearization property allows us to refer to these changes as occurring atomically, without interference with other operations. We complete the safety proofs by proving that the coloring of the linked list is always legal.

The algorithm uses three colors, i.e.,  $M = 3$ ; a node is *legally colored* if its color is not equal to the colors of its left and right neighbors. The left and right anchors are legally colored if their color is different from their right or left neighbor, respectively.

Recall that we use a temporary color  $c_0 < c_1$  in two cases during the algorithm (see Algorithm 3): first, a newly inserted node is colored with  $c_0$  until it is assigned its regular color (line ai4); second, prior to removing a node, its right neighbor is colored with  $c_0$  (line ar2), afterwards this neighbor is assigned a regular color (line ar7).

Recall that a node is valid if both its left and right link pointers are not null or it is an anchor. The coloring property of the algorithm is stated in the next lemma:

**Lemma 4** *All valid nodes are legally colored.*

**Proof:** The proof is by induction on the execution order. The base case is when the linked list is empty; in this case the left anchor has  $c_1$  color and the right anchor has  $c_M$  color 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(5), an operation changes a node only if it holds a lock on 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, it is easy to verify by inspecting the code for changing the operation's data set that a remove operation only changes the color of the right node in the data set and that an insert operation only changes the color of the new, i.e. middle, node in the data set. Thus, the following claim can be deduced:

**Claim 5** *An operation changes a node's color only if it holds locks on 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 *InsertRight* operation presented earlier in Figure 3. The data set of the operation,  $op_1$ , is the new node, the source node ( $m_2$ ), and its right neighbor ( $m_3$ ). While  $op_1$  is applying its changes, other operations neither remove nor insert nodes to the left of the source node and to the right of the right neighbor node, and also do not change the colors of the nodes in the data set. Claim 5 imply that other operations do not change the color of an additional consecutive right node ( $m_4$ ). Moreover, by the induction hypothesis, an additional consecutive left node ( $m_1$ ) is legally colored even if its color is changed by another operation, while  $op_1$  is applying its changes. It is left to show that the changes applied by the operation itself leave the nodes legally colored:

Line ai2, update right link of the new node: the new node is not yet valid (Figure 3(b));

Line ai3, update left link of the new node: the new node is valid and it is legally colored (Figure 3(c));

Line ai4, the new node is assigned with a non-temporary color different than its neighbors (line uc6): the new node is legally colored (Figure 3(d));

Line ai5, update left link of the right neighbor ( $m_3$ ): the right neighbor has color different than the colors of the new node and the right consecutive node ( $m_4$ ), and thus it is legally colored (Figure 3(e)).

Line ai6, update right link of the source node ( $m_2$ ): the source node has color different than the colors of the left consecutive node ( $m_1$ ) and the new node, and thus it is legally colored (Figure 3(f));

We next analyze the *Remove* operation presented earlier in Figure 4. The data set of the operation  $op_2$ , is the source node ( $m_3$ ) and its neighbors ( $m_2$  and  $m_4$ ). While  $op_2$  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. Claim 5 implies that other operations do not change the color of an additional consecutive right node ( $m_5$ ). Moreover, by the induction hypothesis, an additional consecutive left node ( $m_1$ ) is legally colored even if its color is changed by another operation, while  $op_1$  is applying its changes. We show again that the operation's changes leave the nodes legally colored:

Line ar2, the right neighbor ( $m_4$ ) is assigned with the temporary color: the right neighbor is legally colored, since the source node ( $m_3$ ) and the right consecutive node ( $m_5$ ) have colors different than the temporary color ( $c_0$ ) (Figure 4(b));

Line ar3, update right link of the left neighbor ( $m_2$ ): the left neighbor has a non-temporary color, different than the color of the consecutive left node ( $m_1$ ), and the temporary color of the right neighbor, and thus it is legally colored (Figure 4(c));

Line ar4, update left link of the right neighbor: the right neighbor is legally colored, since the left neighbor and the right consecutive node have colors different than the temporary color (Figure 4(d));

Line ar5, set right link of the source node to null: the source node is now invalid (Figure 4(e));

Line ar6, set left link of the source node to null: the source node is invalid (Figure 4(e));

Line ar7, the right neighbor is assigned with a non-temporary color different than its neighbors (line uc6): the right neighbor is legally colored (Figure 4(f)).

■

At this point we can get rid of the assumption that arbitrary  $k$ CAS primitives are available. An *InsertRight* operation locks two consecutive nodes, which must have different colors; a *remove* operation locks three consecutive nodes, at most two of which have the same color. Therefore, DCAS suffices for locking all nodes with the same color in an operation's data set, implying the next lemma:

**Lemma 6 (Safety)** *Algorithm DCAS-CHROMO satisfies the sequential specification of doubly-linked list operations.*

## 5.2 Liveness

Once an operation is in its APPLY phase, it is straightforward that if one of its executing processes takes an infinite number of steps the operation completes successfully. It is left to prove that if the operation is blocked in the LOCK phase then other operations complete successfully. We call an iteration of the loop (lines lc4-lc15) within a `lockColor( $c,r$ )` method, where  $c$  is the color the operation tries to lock and  $r$  is the invocation the process is executing, a  *$c$ -locking iteration*. We argue that in every locking iteration of an executing process, whether successful or not, some operation makes progress, ensuring that the algorithm is nonblocking. To prove the locality properties we later argue that in fact, some “nearby” operation makes progress.

Recall that if an operation  $op$  discovers that a node in its data set is inconsistent with its memento due to changes applied by another operation then it fails, and releases all the locks it holds. Thus, an operation releases locks only after it completes successfully or when it fails.

**Lemma 7** Consider an execution interval  $\alpha$ , in which a process  $p$  executes the  $\text{lockColor}(c,r)$  method of an operation  $op_i$ . Between every two consecutive  $c$ -locking iterations applied by  $p$  within  $\alpha$ , some operation either completes successfully or fails.

**Proof:** The first  $c$ -locking iteration of  $p$  in  $\alpha$  fails since another operation  $op_j$  is blocking  $op_i$ , i.e.,  $op_j$  holds the lock of a node  $nd$  with color  $c$  that  $op_i$  is trying to lock. While executing the  $\text{lockColor}(c,r)$  method  $p$  applies another locking iteration, which means  $op_i$  did not fail, and  $p$  invokes the  $\text{helpBlocking}$  method (line lc15) in the first locking iteration. If  $p$  discovers that  $op_j$  still holds the lock (line hb2) then it helps  $op_j$  (line hb4) before it starts the second iteration. Otherwise,  $op_j$  no longer holds the lock of  $nd$ . In both cases  $op_j$  releases its locks, which means it either completes successfully or it fails. ■

In order to prove the algorithm is nonblocking we define several counters whose increase imply some progress in the execution:

- A *completed operations counter* denoted  $co$ , a nondecreasing counter, that is initially set to 0; increases each time an operation completes successfully.
- A set of  $n$  *failure counters* denoted  $f_1, \dots, f_n$ , one nondecreasing counter per process, that are initially set to 0; failure counter  $f_i$  increases whenever the operation of process  $p_i$  fails.
- A set of  $n$  *color counters* denoted  $cl_1, \dots, cl_n$ , one counter per process,  $cl_i$  holds the color of the last locking iteration that process  $p_i$  executed; this counter is initially set when  $p_i$  executes the first locking iteration of its first operation  $op_i^1$ . Note that the color counters are not necessarily monotone.

The value of a counter in a specific configuration  $C$  is denoted as  $co(C)$ ,  $f_i(C)$ , etc.

If a process  $p$  takes an infinite number of steps, it executes infinitely many locking iterations. In such an execution, the initial configurations of all the locking iterations  $p$  executes are denoted  $C_0, C_1, \dots, C_t, \dots$ , in their order of appearance; clearly, these configurations are not consecutive in the execution.

The following lemma argues that each such configuration is a milestone in the progress of the algorithm:

**Lemma 8** Given an execution of a process  $p_k$ , for every  $t > 0$ , at least one of the counters  $co(C_t)$ ,  $f_1(C_t), \dots, f_n(C_t)$  or  $cl_k(C_t)$  is (strictly) greater than the corresponding counter in configuration  $C_{t-1}$ .

**Proof:** Assume that at  $C_{t-1}$   $p_k$  starts a  $c$ -locking iteration while executing a  $\text{lockColor}(c,r)$  method of the operation  $op_i^j$ , and that at  $C_t$   $p_k$  starts a  $c'$ -locking iteration while executing a  $\text{lockColor}(c',r')$  method of the operation  $op_{i'}^{j'}$ . The proof is by a case analysis.

First assume that  $i = i'$ , i.e., both operations are of the same process  $p_i$ . If  $j < j'$ ,  $op_i^j$  completed successfully and the completed operations counter is incremented, showing that  $co(C_{t-1}) < co(C_t)$ . Otherwise,  $j = j'$ , i.e., both locking iterations are of the same operation. If  $r < r'$ , then  $op_i^j$  failed at the first iteration and thus  $f_i(C_{t-1}) < f_i(C_t)$ . Otherwise,  $r = r'$  i.e., both locking iterations are of the same invocation. In this case the colors  $p_i$  is locking are nondecreasing: If  $c < c'$  then  $cl_k(C_{t-1}) < cl_k(C_t)$ ; otherwise,  $c = c'$  and by Lemma 7 some operation  $op_{i''}$  either completed successfully or failed, hence  $co(C_{t-1}) < co(C_t)$  or  $f_{i''}(C_{t-1}) < f_{i''}(C_t)$ .

Alternatively  $i \neq i'$ , and there are two possible scenarios. In the first scenario,  $op_i$  succeeded locking  $c$  in the first iteration. Since the next locking iteration is of another operation,  $op_i$  locked all the nodes in its data set and  $p_k$  helps it to complete successfully, thus  $co(C_{t-1}) < co(C_t)$ . In the second scenario,  $op_i$  failed locking  $c$  since  $op_{i'}$  blocked it and when  $p_k$  executes the second locking iteration it helps  $op_{i'}$  to lock a higher color, thus  $cl_k(C_{t-1}) < cl_k(C_t)$ . ■

An operation  $op$  fails only due to a change applied by another *failing* operation that holds a lock on a node in  $op$ 's data set.

**Lemma 9** *Algorithm DCAS-CHROMO is nonblocking.*

**Proof:** We need to prove that some pending operation completes after a finite number of  $p_i$ 's steps. By Lemma 8, at least one counter increases at each locking iteration  $p_i$  executes. The color counter,  $cl_i$ , is bounded by  $M$ , the number of colors. Hence, after at most  $M$  consecutive locking iteration, some counter other than the color counter must increase. If the completed operations counter increases, then some pending operation completes. Otherwise some failure counter increases.

We now show that the number of iterations in which a failure counter  $f_k$  increases before some operation completes is bounded by a finite number. As long as a failing operation is not completed, it holds the lock on a node in  $op_k$ 's data set. Thus, the number of concurrent failing operations that are pending can be at most as the size of the operation's data set, in our case, 3. Since once an operation is applying its changes it can not fail, and the number of changes each operation applies before it completes successfully is constant, after an executing process of  $op_k$  executes a finite number of locking iterations, one of  $op_k$ 's failing operations completes. ■

### 5.3 Locality Properties

The locality properties of the algorithm depend on an orientation of the conflict graph that is induced by the order in which nodes in the data set are locked. This captures the locking order of the recursive helping.

The *wait-for graph* of a reachable configuration  $C$  is a directed graph denoted  $H(C)$ .  $H(C)$  is an orientation of  $G(C)$ , including only edges connecting a locked and an unlocked vertices in the data set. Formally,  $H(C)$  contains an edge  $m_v \xrightarrow{op_i, r} m_u$  if and only if  $m_v$  is locked by an operation  $op_i$  in its  $r$ -th LOCK phase, while  $m_u$  is not yet locked by the operation in configuration  $C$ . We denote an edge labeled  $op, r$  by  $e_{op, r}^r$ , or  $e_{op}$  if the sequence number of the invocation is of no importance. For example, consider the following execution of the operations in Figure 1(a): In configuration  $C_1$ ,  $op_2$  locks  $m_3$  (colored *red*); later in configuration  $C_2$   $op_4$  locks  $m_6$  (colored *red*);  $op_3$  locks  $m_4$  (colored *green*) in configuration  $C_3$ ; and  $op_4$  locks  $m_7$  and  $m_5$  (colored *green* and *blue* respectively) in configuration  $C_4$ . Figures 6(a), 6(b), 6(c) depict the respective wait-for graphs of configurations  $C_2$ ,  $C_3$  and  $C_4$ . Figure 6(d) presents another possible execution in which  $op_2$  succeeds to acquire the lock of  $m_4$  prior to  $op_3$ , in some configuration  $C'_2$  following configuration  $C_1$ .

A *wait-for path* from  $op_i$  to  $op_j$  is a directed path from a vertex in  $\mathcal{DS}(op_i)$  to a vertex in  $\mathcal{DS}(op_j)$ , where each edge in the path is in some wait-for graph  $H(C)$ ,  $C \in I_{\{op_i, op_j\}}$ . It is possible that the path does not appear in the wait-for graph of a single configuration  $C$ . A path with one vertex is an *empty wait-for path*.

An operation  $op$  *traverses a wait-for path* of length  $l$ , if its initiator process  $p$  recursively invokes  $l + 1$  **help** methods, within the context of its **help** method (denoted the 0-th recursive call): the first  $l$  recursive calls correspond to the edges of the path. That is, if the  $i$ -th edge,  $1 \leq i \leq l$ , is  $e_{op_{\gamma_i}}^r = \langle m_{i-1}, m_i \rangle$  then the  $i$ -th recursive call is the **help**( $r_i$ ) method of operation  $op_{\gamma_i}$ , and it is invoked after failing to lock  $m_{i-1}$  in the  $i - 1$ -th recursive call; the last recursive call is of the operation owning the lock on the last vertex in the path  $m_l$ , as was read by  $p$  in the penultimate **help** method.

The next lemma argues that every edge traversed by a process in a wait-for path increases the color of the locking iteration that the process executes.

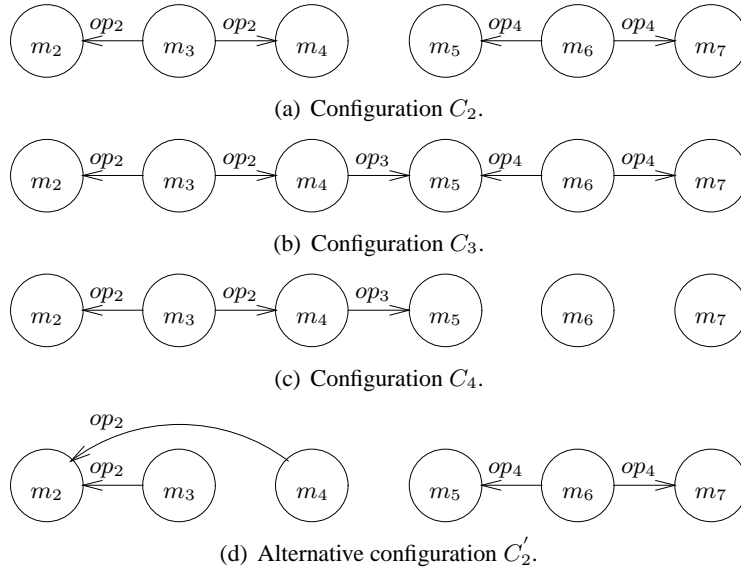


Figure 6: Simple wait-for graphs.

**Lemma 10** *If the initiator of operation  $op_i$  invokes the  $\text{help}(r_j)$  method of an operation  $op_j$  after traversing some wait-for path from  $op_i$  to  $op_j$  of length  $k \geq 0$ , then  $op_j$  owns the lock of a node with color  $c_l \geq c_{k+1}$  in its  $r_j$ -th invocation.*

**Proof:** The proof is by induction on the length of the wait-for path. The base case is when  $k = 0$ ; in this case,  $op_i$  fails to lock some node  $nd$  and discovers  $nd$  is locked by  $op_j$  in the  $r_j$ -th invocation. When  $op_j$  locks  $nd$ , it is a valid node with a non-temporary color  $c_l$ , thus  $op_j$  already locked color  $c_l \geq c_1$  in its  $r_j$ -th invocation.

For the induction step, consider a wait-for path of length  $k > 0$ ;  $p_i$  traverses a wait-for path of length  $k - 1$ , in the last step while executing the  $\text{help}(r_k)$  method of  $op_k$  it tries to lock some node  $t$ . By the induction assumption,  $op_k$  already locked a color greater or equal to  $c_k$  in the  $r_k$ -th invocation, which means  $t$  has color  $c_l > c_k$ .

We review  $p_i$ 's steps while traversing the last wait-for edge  $e_{op_k}^{r_k} = \langle s, t \rangle$ :  $p_i$  reads  $t$ 's lock (line lc10), and discovers  $op_k$  failed to lock  $t$  (otherwise it returns in line lc11) and that  $t$  is consistent with its memento (line lc12), i.e., its color is still  $c_l$ . Then  $p_i$  retrieves the information about the blocking operation,  $op_j$  and its invocation number  $r_j$  from  $t$ 's lock (line hb3), and helps  $op_j$ ; i.e., invokes its  $\text{help}(r_j)$  method (line hb4). Thus, the blocking operation,  $op_j$ , owns the lock of a node with color  $c_l \geq c_{k+1}$  in its  $r_j$ -th invocation. ■

The  $d$ -neighborhood of an operation interval  $I_{op}$ , denoted  $\mathcal{N}_d(I_{op})$ , is the set of operations within distance  $d$  from  $op$  in the conflict graph of  $I_{op}$ . We complete by proving that two operations at constant distance from each other do not interfere with each other's executions:

**Lemma 11 (Local step complexity)** *Algorithm DCAS-CHROMO has 2-local step complexity.*

**Proof:** The step complexity of an operation  $op$  is linear in the number of locking iterations executed by its initiator during  $I_{op}$ . We associate every such locking iteration with some operation. A successful locking iterations is associated with the operation that requested the lock, and to which the process is helping.

Futile locking iterations, however, are associated with the operation that (implicitly) caused the locking to fail, as explained below.

First we prove that a process  $p$  only helps operation in  $\mathcal{N}_2(I_{op})$ , this implies that all operations associated with successful locking iterations are in  $\mathcal{N}_2(I_{op})$ . Since the algorithm uses only 3-coloring, Lemma 10 implies that if  $p$  invokes the  $\text{help}(r_i)$  method of operation  $op_i$  then it traversed some wait-for path from  $op$  to  $op_i$  of length at most  $M - 1 = 2$ . Since all edges traversed by  $op$  are in the conflict graph of  $I_{op}$ ,  $op_i$  is in  $\mathcal{N}_2(I_{op})$ .

Next we prove that an operation that is associated with an unsuccessful locking iteration is in  $\mathcal{N}_2(I_{op})$ . The proof depends on the length of the wait-for path traversed by  $p$ . Assume  $p$  helps an operation  $op_{i_0}$  after traversing a wait-for path of length 0, i.e.,  $op_{i_0} \in \mathcal{N}_0(I_{op})$ .  $p$  fails to lock a node  $nd$  in a  $c$ -locking iteration since another operation  $op_{i_1} \in \mathcal{N}_0(I_{op_{i_0}})$  locked  $nd$ . If this invocation of  $op_{i_0}$  fails then  $op_{i_1}$  changed  $nd$  since it was cloned and thus  $op_{i_1}$  is responsible for the failure of the locking iteration.  $op_{i_1} \in \mathcal{N}_1(I_{op})$  and  $\mathcal{N}_1(I_{op}) \subseteq \mathcal{N}_2(I_{op})$  so  $op_{i_1} \in \mathcal{N}_2(I_{op})$ . Otherwise, the invocation does not fail, this means  $op_{i_1}$  released the node without applying further changes to it. If  $op_{i_1}$  released the node after it completed successfully then it is the responsible,  $op_{i_1} \in \mathcal{N}_1(I_{op}) \subseteq \mathcal{N}_2(I_{op})$ . If  $op_{i_1}$  released the node after it fails, then another operation  $op_{i_2} \in \mathcal{N}_0(I_{op_{i_1}})$  changed a node in  $op_{i_1}$ 's data set and  $op_{i_2}$  is the responsible for the failure of the locking iteration,  $op_{i_2} \in \mathcal{N}_2(I_{op})$ .

Now, assume  $p$  helps an operation  $op_{i_1}$  after traversing a wait-for path of length 1, i.e.,  $op_{i_1} \in \mathcal{N}_1(I_{op})$ . By Lemma 10,  $op_{i_1}$  holds the lock of a node with color  $c_1 \geq c_2$ . This means  $p$  fails to lock a node  $nd$  in a  $c_3$ -locking iteration since another operation  $op_{i_2} \in \mathcal{N}_0(I_{op_{i_1}})$  locked  $nd$ . Since  $op_{i_2}$  already locked color  $c_3$  it completed locking all nodes in its data set and it cannot fail. Thus,  $op_{i_2}$  is responsible for the failure of the locking iteration, and here  $op_{i_2} \in \mathcal{N}_2(I_{op})$ .

Finally, assume  $p$  helps an operation  $op_{i_2}$  after traversing a wait-for path of length 2, i.e.,  $op_{i_2} \in \mathcal{N}_2(I_{op})$ . By Lemma 10,  $op_{i_2}$  already locked color  $c_3$ ; hence, it completed locking all nodes in its data set and  $p$  does not execute any locking iterations while helping it. ■

**Lemma 12 (Local contention)** *Algorithm DCAS-CHROMO has 6-local contention complexity.*

**Proof:** When an operation  $op_i$  helps another operation  $op_k$ , it accesses  $op_k$ 's nodes and operation object. We consider the general case in which two operations  $op_i \neq op_j$  help a third operation  $op_k$  (which may be either  $op_i$  or  $op_j$  themselves), and access the same object simultaneously. By similar arguments to those applied in the previous proof,  $op_i$  (respectively,  $op_j$ ), only helps operations in  $\mathcal{N}_2(I_{op_i})$  (respectively,  $\mathcal{N}_2(I_{op_j})$ ).  $op_i$  and  $op_j$  access the same object, while helping  $op_k$ . So, in the conflict graph of  $\{op_i, op_j\}$  there are edges of wait-for paths, at most of length 2, from  $op_i$  and  $op_j$  to  $op_k$ , these paths end in two nodes in  $\mathcal{DS}(op_k)$ , which are either the same or are both connected with a single edge to the *source* node of  $op_k$ . Thus, the distance between  $op_i$  and  $op_j$  in the conflict graph of  $\{op_i, op_j\}$  is at most 6. ■

## 6 CAS-Based Priority List and Deque Algorithms

When removals occur only at the ends of the linked list, Algorithm DCAS-CHROMO can be further adapted to use only CAS.

Without DCAS, we can no longer atomically lock all nodes with the same color. This is not a problem for insertions—locking only two consecutive nodes on the list, which must have distinct colors. A remove operation still needs to lock three items, two of which may have the same color, but one of the items is always an anchor of the linked list. This inherent asymmetry of the data set is exploited in order to avoid

---

**Algorithm 5** Algorithm CAS-CHROMO: Modification for CAS-based operations

---

```
class InsertFirst : Operation {
  // source : Node = left anchor
  newNode : Node
}
lnf1: Operation::lockTwoNodes(..) {
  // same arguments as in Algorithm 2
  // nd1 is to the left of nd2
lnf2:   lockOneNode(nd1,lock1,seq)
lnf3:   lockOneNode(nd2,lock2,seq)
lnf4: }

RemoveFirst::cloneDataset() {
  cloneNode(source, 0)
  first ← datasetMemento[0].right.node
  cloneNode(first, 1)
  second ← datasetMemento[1].right.node
  cloneNode(second, 2)
}

Operation::toUnlockEmptyState() {
  s ← state
  if s != ⟨seq,APPLY,SUCCESS⟩ then return
  CAS(state, s, ⟨seq,UNLOCK,EMPTY⟩)
}

class RemoveFirst : Operation {
  // source : Node = left anchor
  popped : Node
}
tl1: Operation::touchLocks(Node nd1,Node nd2) {
tl2:   l1, l2 ← nd1.lock, nd2.lock
tl3:   CAS(nd1.lock, l1, ⟨l1.lock,l1.seq,l1.aba1+1⟩)
tl4:   CAS(nd1.lock, l2, ⟨l2.lock,l2.seq,l2.aba1+1⟩)
tl5: }

RemoveFirst::applyChanges() {
  if datasetMemento[2] is ⊥ then
    toUnlockEmptyState() // Empty linked list
  return
  nd ← datasetMemento[1].node
  CAS(popped, null, nd)
  setTempColor(2)
  updateRight(0,2)
  updateLeft(2,0)
  updateRight(1,⊥)
  updateLeft(1,⊥)
  updateColor(2)
}
```

---

waiting circles in the conflict graph, i.e., by locking nodes with the same color in the same direction, from left to right.

We reuse the core implementation of operations from Algorithm DCAS-CHROMO and add the operations *InsertFirst*, *InsertLast*, *RemoveFirst*, and *RemoveLast* for manipulating the ends of the linked list, with the obvious functionality. We discuss the operations applied on the left end of the linked list; the two operations on the right end are symmetric.

Algorithm 5 presents the main changes in the pseudo code. *InsertFirst* and *RemoveFirst* operations are closely related to the *InsertRight* and *Remove* operations, except that they implicitly take the left anchor as their source node. Moreover, the *RemoveFirst* operation refines the methods for cloning and changing the data set. If the list is empty, no changes are applied and the result of the operation is set to EMPTY. Otherwise, the first node is removed from the list after its reference is assigned to the *popped* node attribute.

The most crucial modification is in the locking protocol, which no longer uses a DCAS primitive when locking its data set (line lnf1). A direct consequence is that the process no longer atomically locks nodes with the same color. This requires further adjustments to the proofs, which we describe next.

First we review safety. The fact that equally colored nodes are locked atomically is referenced only in the proof of Lemma 2. Lemma 2(1) claims that a node  $nd$  cannot be locked by operation  $op$  after a transition to UNLOCK phase due to a change in node  $nd'$  in  $op$ 's data set memento. The original proof first handles

the case where  $nd$  and  $nd'$  have the same color. We slightly modify the `lockColor` method: after a change in some node is detected (line lc6 or line lc12) and before a transition to the UNLOCK phase occurs (line lc7 or line lc13), the executing process calls the `touchLocks` method (line tl1), which essentially violates the locks by “touching” their ABA-prevention counter.<sup>2</sup>

Given this change, the proof can be fixed. An executing process of  $op$ ,  $p_i$ , verifies that  $nd$  and  $nd'$  are valid and has not changed. If  $nd'$  changed, it must be that another executing process,  $p_j$ , “touched” the locks of  $nd$  and  $nd'$ , and then  $op$  shifts to UNLOCK phase. Since at least the ABA-prevention counter of  $nd$ 's lock is changed before the transition, any attempt of  $p_i$  to lock  $nd$  after the transition, fails. The proofs of the other cases, where the colors of  $nd$  and  $nd'$  are different, do not rely on the atomicity of locking equally-colored nodes, and they remain the same. Likewise, the proofs of Lemma 2(2)-(6) and other safety lemmas are the same as for Algorithm DCAS-CHROMO.

Next we discuss liveness. Lemma 8 claims that the value of at least one counter increases at every locking iteration. The last claim in the proof is based on the fact the an operation  $op_i$  helps another operation  $op_j$  only if it already locked a higher color. In Algorithm CAS-CHROMO the color counter may not increase if the process helps a remove operation. Nevertheless, there can be at most two remove operations concurrently, which lock nodes with the same color according to their order in the list, from left to right. Thus, after at most three locking iterations in which the counters remain the same, the value of one counter increases. So, the proof claiming that after a finite number of locking iterations some operation completes is still valid, and the algorithm is nonblocking.

Finally, to discuss the locality properties of the algorithm, we revisit Lemma 10. This lemma states that if a process helps an operation  $op$  after traversing a wait-for path of length  $k$ , then  $op$  already locked color  $c_l \geq c_{k+1}$ . The proof is based on the fact that the color increases with each step on the path. In Algorithm CAS-CHROMO the color may not increase when traversing one edge of a remove operation in the path, but then again there can be at most two remove operations concurrently. So, the lemma now claims that  $op$  already locked color  $c_l \geq \max\{c_{k-1}, c_1\}$  since the colors in the path increase at each step, excluding at most two non-decreasing edges. The revised lemma, and restrictions induced by the structure of a doubly-linked list, imply the following theorem:

**Theorem 13** *Algorithm CAS-CHROMO is a nonblocking implementation of a priority queue with 4-local step complexity and 4-local contention complexity.*

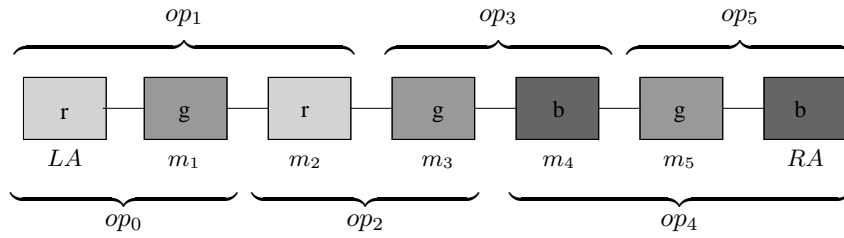
**Sketch of proof:** The proof of the local step complexity property closely follows the arguments in the proof of Lemma 11. Due to the modification in Lemma 10, we replace the 2-neighborhood of an operation interval  $I_{op}$  in 4-neighborhood,  $\mathcal{N}_4(I_{op})$ , and prove the algorithm has 4-local step complexity.

For the proof of the local contention property it can be shown that the worst-case scenario is the one presented in Figure 7. Assume  $op_0$ ,  $op_2$ ,  $op_3$  and  $op_5$  are *InsertRight* operations,  $op_1$  is a *RemoveFirst* operation and  $op_4$  is a *RemoveLast* operation. Consider an execution in which  $op_1$  locks the left anchor,  $op_2$  locks  $m_2$ ,  $op_3$  locks  $m_3$ ,  $op_4$  locks  $m_4$ , and  $op_5$  locks  $m_5$  and the right anchor.  $op_0$  and  $op_5$  are at distance four from each other, and they contend while accessing the right anchor:  $op_0$  tries to lock the left anchor and thus it helps  $op_1$ ;  $op_1$  tries to lock  $m_2$  and thus it helps  $op_2$ ;  $op_2$  tries to lock  $m_3$  and thus it helps  $op_3$ ;  $op_3$  tries to lock  $m_4$  and thus it helps  $op_4$ ;  $op_4$  tries to lock the right anchor and thus it helps  $op_5$ . ■

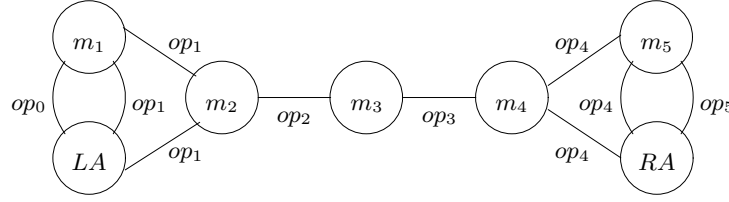
An implementation of the deque data structure requires operations only at the ends. In this case the analysis can be further tightened to show:

---

<sup>2</sup>This is similar to the way an operation is invalidated before releasing its nodes in [4].



(a) Overlapping deque operations.  $LA$  and  $RA$  are left and right anchors respectively



(b) The corresponding conflict graph  $G(C)$ .

Figure 7: Algorithm CAS-CHROMO worst case scenario.

**Theorem 14** *Algorithm CAS-CHROMO is a nonblocking implementation of a deque with 2-local step complexity and 2-local contention complexity.*

**Sketch of proof:** We prove that two operations are either at distance smaller than or equal to 2, or their distance is infinite. For any operation  $op$  in a given conflict graph, we can compute the distance to any other operation in the graph by applying a breadth first order from  $op$ 's data set vertices, denoted the zero layer. We say a layer in the search *covers* an operation if the operation's data set is covered by the layer. Assume without loss of generality that  $op$  is an operation on the first end of the linked list. The first layer of the search covers all the operations on the first end of the list. If there is an operation on the last end of the linked list, which includes data items from the first layer in its data set then it is covered in the second layer, and the third layer covers all operation on the last end which are not covered by the second layer. Operation covered by the first, second and third layers are at distance zero, one and two respectively from  $op$ . Thus, either the distance between two operation is  $\infty$ , in which case they do not contend or affect each others step complexity, or they are at distance smaller or equal to two. ■

## 7 Discussion

This paper presents a new approach for designing nonblocking and high-throughput implementations of linked list data structures; our scheme may have other applications, e.g., for tree-based 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 is modified to use only CAS. These implementations are intended as a proof-of-concept and leave open further optimizations that will 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.

**Acknowledgments:** We thank David Hay, Danny Hendler, Gadi Taubenfeld and the referees for helpful comments.

## References

- [1] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *PODC 1997*, pages 111–120.
- [2] O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. L. Steele Jr. DCAS-based concurrent dequeues. *Theory Comput. Syst.*, 35(3):349–386, 2002.
- [3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.
- [4] H. Attiya and E. Dagan. Improved implementations of binary universal operations. *J. ACM*, 48(5):1013–1037, 2001.
- [5] H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, second edition, 2004.
- [6] G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA 1993*, pages 261–270.
- [7] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele Jr. Even better DCAS-based concurrent dequeues. In *International Symposium on Distributed Computing*, pages 59–73, 2000.
- [8] S. Doherty, D. Detlefs, L. Grove, C. H. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA 2004*, pages 216–224.
- [9] F. E. Fich, V. Luchangco, M. Moir, and Nir Shavit. Obstruction-free step complexity: Lock-free dcas as an example (brief announcement). In *DISC 2005*, pages 493–494.
- [10] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *PODC 2002*, pages 260–269.
- [11] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
- [12] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*, pages 388–402.
- [13] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC 2001*, pages 300–314.
- [14] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
- [15] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS 2003*, pages 522–529.
- [16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [17] IBM. *IBM System/370 Extended Architecture, Principle of Operation*, 1983. IBM Publication No. SA22-7085.
- [18] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC 1994*, pages 151–160.
- [19] M. M. Michael. CAS-based lock-free algorithm for shared dequeues. In *Euro-Par 2003*, pages 651–660.
- [20] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA 2002*, pages 73–82.
- [21] N. Shavit and D. Touitou. Software transactional memory. *Dist. Comp.*, 10(2):99–116, 1997.
- [22] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.
- [23] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based dequeues using single-word compare-and-swap. In *OPDIS 2004*, pages 240–255.
- [24] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *PODS 1992*, pages 212–222.