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

(Extended Abstract)

Hagit Attiya and Eshcar Hillel

Department of Computer Science, Technion

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 simpli£es 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 bene£ts 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.

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 identi£ers 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 [15], 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 signi£cant drawbacks: Some implementations may contain garbage nodes [14], others statically limit the data structure's size [16] or do not allow concurrent operations on both ends of the queue [21]. Even when DCAS (i.e., 2CAS) is used, existing implementations either are inherently sequential [11, 12] or allow to access chains of garbage nodes [9].

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. [9]).

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* [22] or the so-called *locking without blocking* techniques [7, 25]. 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* [15], namely, in any in£nite execution, some pending operation completes within a £nite 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.

To evaluate whether operations that access disjoint parts of the data structure, or are widely separated in time, do not interfere with each other, Afek et al. [1] de£ne two measures. These de£nitions rely on the familiar notion of a *con¤ict 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 con¤ict 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 con¤ict graph can delay each other; it has *d-local contention* if only operations in distance less than or equal to d in the con¤ict graph can access the same locations simultaneously.[1] In particular, when there is no path in the con¤ict graph between the data items 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* [19].

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-

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

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 speci£c 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 £rst. In return, the operation needs to guarantee that the modi£cations 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 re£ned 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 dif£cult: 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 £rst 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* [10]; namely, an operation completes within $O(1)$ steps in an execution suf£x 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 ¤urry of papers proposed implementations of dynamic linked list data structures, yet none of them provided all the features of our algorithms.

Harris [14] used CAS to implement a singly-linked list, with insertions and removals anywhere; however, in this algorithm, a process can access a node previously removed from the linked list, possibly yielding an unbounded chain of uncollected garbage nodes. Michael [20] handled these memory management issues. Elsewhere [21], 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 [24] 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 [23]; in the latter algorithm, a long path of overlapping removals may cause interference among distant operations; moreover, during intermediate states, there can be a consecutive sequence of inconsistent backward links, causing part of the list to behave as singly-linked. An *obstruction-free* deque, providing a liveness property weaker than nonblocking, was proposed by Herlihy et al. [16]; besides blocking when there is even a little contention, this array-based implementation bounds the deque's size.

Greenwald [11, 12] 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 £rst 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 [8] is an attempt for further improvement that uses only a single DCAS primitive per operation in the best case, instead of two. Unfortunately, SNARK is incorrect [9]; 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. [9] even argue that primitives more powerful than DCAS, e.g., 3CAS, are needed in order to obtain simple and ef£cient 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 de£nes 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 modi£cations needed to obtain the CAS-based implementation that does not allow removals from the middle. The complete code and proof of correctness for both algorithms appear in the full version of this paper [5].

2 Preliminaries

We consider a standard model for a shared memory system [6] in which a £nite set of *asynchronous processes* $p_1, \ldots, p_n$ communicate by applying *primitive* operations to m shared *memory locations*, $l_1, \ldots, l_m$.

A *con£guration* is a vector $C = (q_1, \ldots, q_n, v_1, \ldots, v_m)$, 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 allow the following primitives: READ(l_j) returns the value v_j in location l_j; WRITE(l_j, v) sets the value of location l_j to v; CAS(l_j, exp, new) writes the value *new* to location l_j if its value is equal to *exp*, and returns a success or failure ¤ag; DCAS is similar to CAS, but operates on two independent memory locations.

An *execution interval* α is a (£nite or in£nite) alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \ldots$, where C_k is a con£guration, ϕ_k is an event and the applica-

tion of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \ldots$. An *execution* is an execution interval in which C_0 is the unique initial con£guration.

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 speci£cation*, which indicates how it is modi£ed when operations are applied in a serial manner (in isolation).

An *implementation* of a data structure T provides a speci£c data-representation for T's instances as a set of memory locations, and protocols that processes must follow to carry out T's operations, de£ned in terms of primitives applied to memory locations. We require the implementation to be *linearizable* [17].

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 £rst (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 con£guration 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 follows 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.

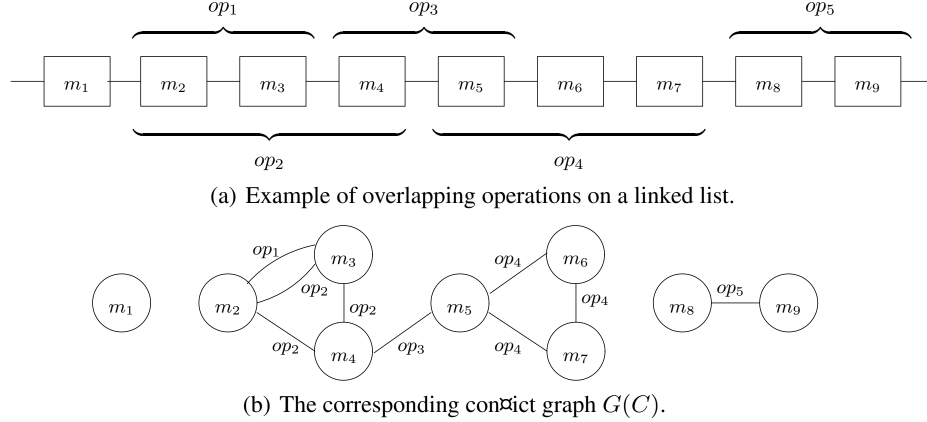
remove() If $source$ is a valid node other than an anchor, then remove $source$ from the linked list and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

In order to apply an operation op_i to the data structure, process p_i executes the associated protocol. The *interval of an operation op*, denoted I_{op}, is the execution interval between the £rst and last events of the process executing *op*'s protocol; if the operation does not terminate, its interval is in£nite. 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. The lock set of an operation op_i applied in state s is denoted $\mathcal{LS}_s(op_i)$. 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 concurrent, the state of the data structure at a con£guration C is not necessarily unique. A state s of the data structure is *possible* in con£guration C,



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

(b) The corresponding con¤ict graph $G(C)$.

Fig. 1. A simple con¤ict graph.

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 *state*(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 need to 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 con£guration C is de£ned as $\mathcal{DS}_C(op_i) = \bigcup_{s \in state(C)} \mathcal{LS}_s(op_i)$, 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 $state(C)$. $\mathcal{DS}(op_i) = \bigcup_{C \in I_{op_i}} \mathcal{DS}_C(op_i)$; namely, the union of $\mathcal{DS}_C(op_i)$ over all con£gurations during op_i's execution interval.

The *con¤ict graph* of a con£guration C, denoted $G(C)$, occurring in some execution, is an undirected graph that captures the distance between overlapping operations. If C is in the execution interval of an operation op_i, and v and u are data items in $\mathcal{DS}_C(op_i)$, then the con¤ict graph includes an edge between the respective vertices m_v and m_u, labeled op_i. The con¤ict graph of an execution interval α is the graph $\bigcup_{C \in \alpha} G(C)$. For example, Figure 1(a) depicts the data set of 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. Figure 1(b) depicts the corresponding con¤ict graph; the new node, omitted from the £gure, is also in the operation's data set.

The *con¤ict distance* (in short *distance*) between two operations, op_i, op_j, in a con¤ict graph is the length (in edges) of the shortest path between some vertex m_i in $\mathcal{DS}(op_i)$ and some (possibly the same) vertex m_j in $\mathcal{DS}(op_j)$. In particular, if $\mathcal{DS}(op_i)$ intersect $\mathcal{DS}(op_j)$, then the distance between op_i and op_j is zero. The distance is ∞, if there is no such path. In the con¤ict graph of 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 ∞.

We use this dynamic version of a con¤ict graph in the de£nitions of locality measures suggested by Afek et al. [1]:

De£nition 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 con¤ict graph of its operation interval* I_{op}.

De£nition 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 con¤ict graph of* $I_{\{op_1,op_2\}}$ *is smaller than or equal to* d.

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 £rst review this scheme, and then describe our algorithm.

The Color-Based Virtual Locking Scheme: Data structures can be implemented by the nonblocking *virtual locking* scheme [7,22,25]. 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); £nally, 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.

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 α 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 α's con¤ict graph, from some vertex in $\mathcal{DS}(op_1)$. In general, op_1 can be delayed by any operation within £nite distance from it, implying that the locality is high.

Shavit and Touitou [22] overcome this problem by helping only an immediate neighbor in the con¤ict graph. Nevertheless, the number of steps a process performs depends on the length of the longest path from its data set in the con¤ict graph. Consider again

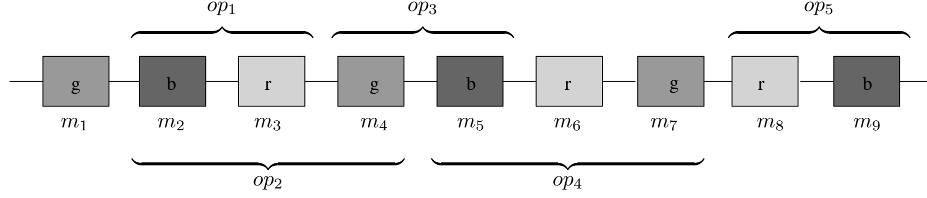


Fig. 2. 3-Coloring of the linked list in Figure 1(a).

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, $c_1 < c_2 < \ldots < c_M$. 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 $r(red) < g(green) < b(blue)$. 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 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.

Our Approach: We achieve constant locality properties by employing two complementary algorithmic ideas: The £rst 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: $b, r, b, r, b, r, \ldots$. Assume a set of concurrent operations, each of which is trying to remove a different r-colored node, by £rst locking the node and its two b-colored neighbors. An implementation that locks these two b-colored nodes one at a time, e.g., £rst the left neighbor, can lead to a con£guration 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 *£ve* consecutive 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 suf£ces 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 [22] 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 [13], using a *data set memento*, which holds a view of the data set when the operation starts. If, while locking, a node and its memento are inconsistent, 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.

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.

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:

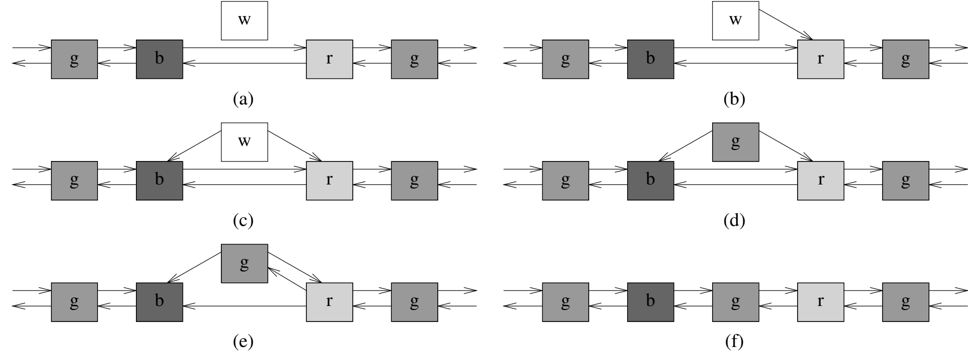


Fig. 3. An example of an *InsertRight* operation - op_1 in Figure 2

```
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 £gures, 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 neighbor of m (Figure 3(b)); update left neighbor of m—now, m is legally colored, since its neighbors m_2 and m_3 have colors different than w (Figure 3(c)); m is assigned with a non-temporary color different than its neighbors m_2 and m_3 (Figure 3(d)); update left neighbor of m_3 (Figure 3(e)); update right neighbor of m_2 (Figure 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 neighbor of m_2 (Figure 4(c)); update left neighbor of m_4 (Figure 4(d)); set right and left neighbors 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)).

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 middle node, and a *Remove* operation

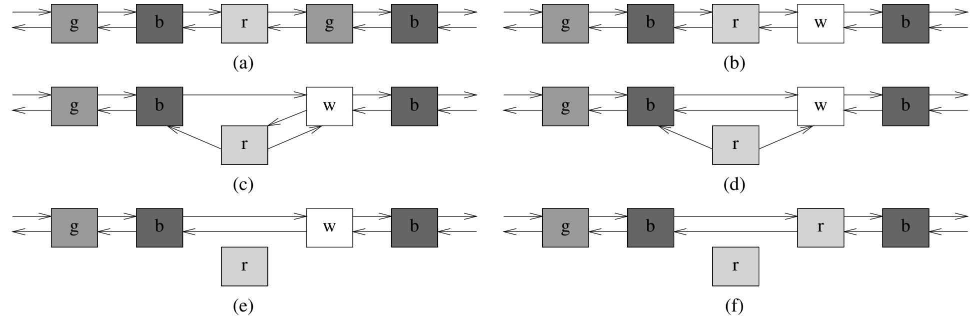


Fig. 4. An example of a *remove* operation - op_2 in Figure 2

changes the color of the right node. The color of the left node in the data set of an operation is not modi£ed. 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*. To initiate an invocation, the initiator process generates the operation's data set memento, which traces inconsistencies in the data set due to changes applied by concurrent operations. If the operation locks its data set and applies its changes then the invocation *completes successfully* and the operation will not be re-invoked. Otherwise, the invocation *fails* and the operation restarts a new invocation.

The state of an operation is a tuple $\langle$seq,phase,result$\rangle$: *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.

Figure 5 shows the state transition diagram of an operation's invocation. The dashed line indicates re-invocation, increasing the sequence number of the operation. The state transitions of an invocation in a best-case execution, encountering no contention, appear at the top. If an operation discovers, while initiating an invocation, that another operation removed the source node then it need not apply its changes, and it skips to the FINAL phase with an INVALID result; this operation will not be re-invoked. If an operation discovers that a node in its data set other than the source node is invalid, then the operation needs to re-evaluate its data set. In such a case, the invocation fails and a new invocation is restarted. Another scenario in which an invocation fails is if the operation detects inconsistency with the data set memento while locking the data set. In this case, the operation releases the locks it already acquired and restarts a new invocation.

When an operation *op* fails to lock color c it may discover that a node in its data set is locked by another, blocking operation op'. In such a case, we follow the standard recursive helping mechanism, i.e., *op* helps op'. Before helping op', the executing process of *op* veri£es (again) that the nodes are consistent with their mementos. This is crucial for maintaining the locality properties of the algorithm. If after an operation fails to lock the nodes it discovers that none of them is locked by another operation, it simply retries to acquire their locks. Finally, when an operation discovers that its source node is in-

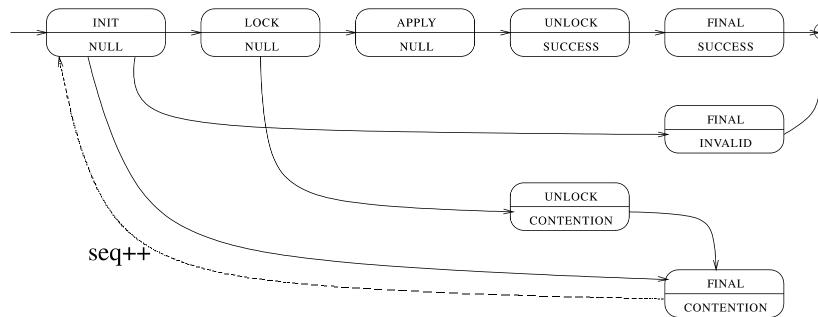


Fig. 5. Diagram of an operation state transitions model; the lower part of the state is the value of *result*.

valid (as described above), it helps the operation that removes this node before skipping to its FINAL phase, to preserve the correct order in which the operations complete.

Since an operation may be invoked more than once, its execution is composed of an alternating sequence of acquiring and releasing locks. Having more than one process executes the operation requires special care. Speci£cally, 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. Before acquiring a lock the process veri£es that the operation's sequence number is equal to the invocation it is executing. Furthermore, to prevent a process from releasing locks acquired in a later invocation, the operation stamps any lock it acquires with its sequence number. Before a process releases a lock, it veri£es that the sequence number stamped on the lock is equal to the invocation it is executing.

Some Implementation Details: We use object-oriented terminology and de£ne operations as objects, whose structure and behavior are de£ned in the *Operation* hierarchy.

A process initializes an operation object with all the data required for its execution, speci£cally the source node from the linked list on which the operation is applied. Algorithm 1 outlines 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 re-invoke the operation (lines ex3-ex4): First it generates the new data set memento (line t5); then it "helps" itself to follow the locking scheme (line t7); lock nodes in its data set (line h2), apply its changes (line h4), and releas the data set (line h6). Concrete operations, such as *InsertRight* and *Remove*, extend the *Operation* structure and re£ne its protocols for cloning and manipulating the data set with respect to their speci£cations. (The full pseudocode appears in [5].)

It is well-known that CAS primitives suffer from the ABA problem [18]: a process p may read a value A from some memory location l, then other processes change l to B and then back to A, later p applies CAS on l and the comparison succeeds whereas it should have failed. The simplest way to avoid this problem is to associate each attribute with a monotonically increasing counter. The attribute and the counter are manipulated atomically; the counter is incremented whenever the attribute is updated. Assuming that the counter has enough bits, the CAS succeeds only if the counter has not changed since

Algorithm 1 Algorithm DCAS-CHROMO: Execution outline

```
ex1: Result Operation::execute() {
ex2:     do
ex3:         initiate new invocation
ex4:         try()
ex5:     while state.result = CONTENTION
ex6:     return state.result
ex7: }

t1: Operation::try() {
t2:     if source is invalid then
t3:         helpBlocking(source.lock)
t4:         transition to FINAL-INVALID state
t5:     clone data set
t6:     transition to LOCK state
t7:     help(state.seq)
t8:     transition to FINAL state
t9: }

h1: Operation::help(int seq) {
h2:     lock data set // by ascending colors
h3:     if state.phase = APPLY then
h4:         apply changes
h5:         transition to UNLOCK state
h6:     unlock data set
h7: }

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

the process read the attribute. Other methods prevent the ABA problem without the use of a per-attribute counters, 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 do not form since the links of removed nodes are nulli£ed. 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, so it is assumed that once a node is removed from one linked list it is not reused until reclaimed by the garbage collector.

5 Correctness Proof (Outline)

The safety properties of the implementation, and in particular, its linearizability, hinge on showing that 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, 7, 22, 25], since the data set is dynamic.

Proving the progress and locality properties is more involved. One key is to show that the color of an item causing a blocked operation to help, increases with every recursive call. This implies that the depth of the recursion is bounded by the number of colors, M. Moreover, we argue that in every locking attempt of an executing process, may it be a successful or a futile one, some "nearby" operation makes progress, ensuring that the algorithm is nonblocking and that the step complexity of an operation depends only on the number of operations in its close neighborhood.

The detailed correctness proof appears in the full version of the paper [5], showing:

Theorem 1. *Algorithm* DCAS-CHROMO *is a nonblocking implementation of a doubly-linked list, allowing insertions and removals anywhere, with 2-local step complexity and 6-local contention complexity.*

6 CAS-Based Doubly-Linked List Algorithm

In this section we discuss Algorithm CAS-CHROMO, a CAS-based variation of Algorithm DCAS-CHROMO, allowing insertions everywhere and removals only at the ends.

We reuse the core implementation of insert and remove operations from Algorithm DCAS-CHROMO and add the operations *InsertFirst*, *RemoveFirst*, *InsertLast* and *RemoveLast* for manipulating the ends of the linked list, with the obvious functionality.

We discuss the operations applied on the £rst (left) end of the linked list; the two operation on the last (right) end are symmetric. *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. The most crucial modi£cation is in the locking protocol, which no longer uses a DCAS primitive when locking its data set. However, nodes with the same color are locked according to their order in the list, from left to right; this allows to prove that the algorithm is nonblocking. In fact, this can also show that operations help only along paths with $O(1)$ length, which can be used to prove that the algorithm has good locality properties. The details of the algorithm, as well as its correctness proof, appear in the full version of the paper [5].

Theorem 2. *Algorithm* CAS-CHROMO *is a nonblocking implementation of a doubly-linked list, allowing insertions anywhere and removals at the ends, with 4-local step complexity and 4-local contention complexity.*

An implementation of deque data structure requires operations only at the ends. In this case, the analysis can be further improved to show that the algorithm has 2-local step complexity and 2-local contention complexity.

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; when nodes are removed only from the ends, the implementation is modi£ed to use only CAS. These implementations are intended only as a proof-of-concept and leave open further optimizations. It is also necessary to implement a *search* mechanism 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. PODC 1997, pp. 111–120..
2. O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS-based concurrent deques. *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 E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. Available from `http://www.cs.technion.ac.il/~hagit/publications/`, 2006.
6. H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations and Advanced Topics*. John Wiley& Sons, second edition, 2004.
7. G. Barnes. A method for implementing lock-free shared-data structures. SPAA 1993, pp. 261–270.
8. D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. DISC 2000, pp. 59–73.
9. S. Doherty, D. Detlefs, L. Grove, C. H. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS is not a silver bullet for nonblocking algorithm design. SPAA 2004, pp. 216–224.
10. F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free step complexity: Lock-free dcas as an example (brief announcement). DISC 2005, pp. 493–494.
11. M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
12. M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. PODC 2002, pp. 260–269.
13. T. Harris and K. Fraser. Language support for lightweight transactions. OOPSLA 2003, pp. 388–402.
14. T. Harris. A pragmatic implementation of non-blocking linked-lists. DISC 2001, pp. 300–314.
15. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
16. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. ICDCS 2003, pp. 522–529.
17. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
18. IBM. *IBM System/370 Extended Architecture, Principle of Operation*, 1983. IBM Publication No. SA22-7085.
19. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. PODC 1994, pp. 151–160.
20. M. Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002, pp. 73–82. ACM Press.
21. M. Michael. CAS-based lock-free algorithm for shared deques. Euro-Par 2003, pp. 651–660.
22. N. Shavit and D. Touitou. Software transactional memory. *Dist. Comp.*, 10(2):99–116, 1997.
23. H. Sundell. *Ef£cient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.
24. H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. OPODIS 2004, pp. 240–255.
25. J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. PODS 1992, pp. 212–222.