Highly-Concurrent Multi-Word Synchronization
(Extended Abstract)

Hagit Attiya and Eshcar Hillel

Department of Computer Science, Technion

Abstract. The design of concurrent data structures is greatly facilitated by the availability of synchronization operations that atomically modify k arbitrary locations, such as *k-read-modify-write* (kRMW). Aiming to increase concurrency in order to exploit the parallelism offered by today's multi-core and multiprocessing architectures, we propose a software implementation of kRMW that efficiently breaks apart delay chains. Our algorithm ensures that two operations delay each other only if they are within distance $O(k)$ in the *conflict graph*, dynamically induced by the operations' data items.
The algorithm uses *double compare-and-swap* (DCAS). When DCAS is not supported by the architecture, the algorithm of Attiya and Dagan [3] can be used to replace DCAS with (unary) CAS, with only a slight increase in the interference among operations.

1 Introduction

Multi-word synchronization operations, like *k-read-modify-write* (kRMW), allow to read the contents of several memory locations, compute new values and write them back, all in one atomic operation. A popular special case is *k-compare-and-swap* (kCAS), where the values read from the memory locations are compared against specified values, and if they all match, the locations are updated. Multi-word synchronization facilitates the design and implementation of concurrent data structures, making it more effective and easier than when using only single-word synchronization operations. For example, removing a node from a doubly-linked list and a right (or left) rotation applied on a node in an AVL tree can easily be implemented with 3CAS and 4CAS, respectively.

Today's multi-core architectures, however, support in hardware only single-word synchronization operations like CAS or at best, *double compare-and-swap* (DCAS). Providing kRMW or kCAS in software has therefore been an important research topic.

It is crucial to allow many operations to make progress concurrently and complete without interference, in order to utilize the capabilities of contemporary architectures. Clearly, when operations need to simultaneously access the same words, an inherent "hot spot" is created and operations must be delayed. A worse and unnecessary situation happens in typical kRMW implementations, when the progress of an operation is hindered also due to operations that do not contend for the same memory words. In

Supported by the *Israel Science Foundation* (grant number 953/06).
The full version of this paper is available from
`http://www.cs.technion.ac.il/~hagit/publications/`.

these implementations [5, 8, 12, 14, 15], an operation tries to lock all the words it needs, one by one; if another operation already holds the lock on a word, the operation is blocked and can either *wait* for the lock to be released (possibly while *helping* the conflicting operation to make progress) or *reset* the conflicting operation and try to acquire the lock.

In these schemes, a *chain* of operations may be created, where each operation in the chain is either waiting for a word locked by the next operation (possibly while helping it), or is being reset by the previous operation in the chain. It is possible to construct *recurring* scenarios where an operation repeatedly waits, helps, or is reset due to each operation along the path. In these scenarios, an operation is delayed a number of steps proportional to the total number of operations in these chains and their length, causing a lot of work to be invested, while only a few operations complete. Evidently, it is necessary to bound the length of the chains, in order to improve the concurrency of a kRMW implementation.

We proceed more precisely, by considering the *conflict graph* of operations that overlap in time; in this graph, vertices represent data items, i.e., memory locations, and edges connect data items if they are accessed by the same operation. The *distance* between two operations in a conflict graph is the length of the path between the operations' data items. Thus, two simultaneous operations contending for a data item have zero distance in the conflict graph. Algorithms of the kind described above guarantee that operations in disconnected parts of the conflict graph do not delay each other; that is, operations proceed in parallel if they access disjoint parts of the data structure; that is, they are *disjoint access parallel* [12].

Even when operations choose their items uniformly at random, it has been shown [7], both analytically and experimentally, that the lengths of such paths depend on the total number of operations, and paths of significant length might be created in the conflict graph. This means that the connected components have non-constant diameter, implying that an operation in the typical multi-word synchronization algorithms can be delayed by "distant" operations, even when an algorithm is disjoint access parallel.

The adverse effect of waiting and delay chains can be mitigated, greatly improving the concurrency, if operations are delayed only due to operations within constant distance. Informally, an implementation is *d-local nonblocking* if whenever an operation *op* takes an infinite number of steps, some operation, within distance d from *op*, completes. This implies that the throughput of the algorithm is localized in components of diameter d in the conflict graph, and they are effectively *isolated* from operations at distance $> d$. This extends the notion of *nonblocking* (also called *lock-free*) algorithms.

Our contribution. We present an algorithm for multi-word synchronization, specifically, kRMW, which is $O(k)$-local nonblocking. The algorithm is flexible and does not fix k across operations. We store a constant amount of information (independent of k), in each data item.

Our main new algorithmic ideas are first explained in the context of a *blocking* implementation (Section 3), in which the failure or delay of an operation may block operations that access *nearby* data items; however, operations that access data items that are farther than $O(k)$ away in the conflict graph are not affected. (This is a slightly weaker property than *failure locality*, suggested by Choy and Singh [6].)

A key insight of our algorithm is that the length of waiting chains can be bounded, yielding better concurrency, if an operation decides whether to wait for another operation or reset it by comparing how advanced they are in obtaining locks on their data items. If the conflicting operation is more advanced, the operation waits; otherwise, the operation resets the conflicting operation and seizes the lock on the item. While a similar approach has been used in many resource allocation algorithms, part of our contribution is in bounding the locality properties of this approach. A particularly intricate part of the proof shows that an operation cannot be repeatedly reset, without some operation in its $O(k)$-neighborhood completing.

Another novelty of our algorithm is in handling the inevitable situation that happens when overlapping operations that has made the same progress, that is, locked the same number of items, create a chain of conflicts. The symmetry inherent in this situation can, in principle, be broken by relying on operation identifiers, so as to avoid deadlocks and guarantee progress. However, this can create delay chains that are as long as the number of operations in this path (which can be n). Instead, we break such ties by having the conflicting operations try to atomically lock the two objects associated with the operations, using *double compare-and-swap* (DCAS). This easily and efficiently partitions the above kind of path into disjoint chains of length 2, ensuring that operations are delayed only due to close-by conflicts.

This scheme is made $3k$-local nonblocking by *helping* a blocking operation that is more advanced, instead of waiting for it to complete; we still reset conflicting operations that are less advanced (see Section 4). In this algorithm, helping chains replace delay chains, which intuitively explains how the $O(k)$ failure locality of the blocking algorithm translates into $O(k)$-local nonblocking. (This intuition is made concrete in the proof of the local nonblocking algorithm.)

Our algorithm demonstrates that DCAS provides critical leverage allowing to implement kRMW, for any $k > 2$, with locality that is difficult, perhaps impossible, to obtain using only CAS. While few architectures provide DCAS in hardware, DCAS is an ideal candidate to be supported by *hardware transactional memory* [10, 13], being a short transaction with static data set of minimal size (two). Alternatively, DCAS can be simulated in software from CAS using the highly-concurrent implementation of Attiya and Dagan [3], which is $O(\log^* n)$-local nonblocking. This yields kRMW implementation from CAS, which is $O(k + \log^* n)$-local nonblocking.

Related work. Afek et al. [1] present a kRMW algorithm, for any fixed k, which can be shown to be $O(k + \log^* n)$-local nonblocking. Their implementation works recursively in k, going through the locations according to their memory addresses, and coloring the items before proceeding to lock them; at the base of the recursion (for $k = 2$), it employs the binary algorithm of Attiya and Dagan [3]. To support the recursion, their implementation stores $O(k)$ information per location. The recursive structure of their algorithm makes it very complicated and infeasible as a basis for practical, dynamic situations, as it requires that k must be hard-wired, uniformly for all operations. In contrast, our algorithm is more flexible, as each operation can access a different numbers of data items. We store a constant amount of information, independent of k. More importantly, our algorithm can be modified not to require all the data items when the operation starts, allowing to extend it to dynamic situations (see Section 5).

Afek et al. [1] define two measures in order to capture the locality of nonblocking algorithm in a more quantitative sense. Roughly, an implementation has *d-local step complexity* if the step complexity of an operation is bounded by a function of the number of operations within distance d of it in the conflict graph; an implementation has *d-local contention* if two operations accessing the same memory location simultaneously are within distance d. Their implementation has $O(k + \log^* n)$-local step complexity and contention,[1] matching the complexities of our algorithm, when the DCAS is implemented as proposed in [3].

The first multi-word algorithms that rely on helping were the "locking without blocking" schemes [5, 15], where operations recursively help other operations, without releasing the items they have acquired. These algorithms are $O(n)$-local nonblocking. The static *software transactional memory* (STM) [14] also provides multi-word synchronization. In this algorithm, operations acquire words in the order of their memory addresses, and help only operations at distance 0; nevertheless, it is $O(n)$-local nonblocking. Harris et al. [8] give an implementation of dynamic multi-word operations with recursive helping, which is $O(n)$-local nonblocking.

2 Preliminaries

We consider a standard model for a shared memory system [4] in which a finite 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 *configuration* is a vector describing the states of processes and the values of memory locations. In the (unique) *initial configuration*, every process is in its initial state and every location contains its initial value.

An *event* is a computation step by a process consisting of some local computation and the application of a primitive to the memory. Besides standard READ and WRITE primitives, we employ CAS(l_j, *exp*, *new*), which writes the value *new* to location l_j if its value is equal to *exp*, and returns a success or failure flag. We also use a DCAS primitive, which is similar to CAS, but operates on two memory locations atomically.

An *execution interval* α is an alternating sequence of configurations and events, where each configuration is obtained from the previous configuration by applying an event. An *execution* is an execution interval starting with the initial configuration.

An *implementation* of a kRMW operation specifies the data representation of operations and data items, and provides algorithms, defined in terms of primitives applied to the memory locations, that processes follow in order to execute operations. The implementation has to be *linearizable* [11].

The *interval of an operation op*, denoted I_{op}, is the execution interval between the first event and last event (if exists) of the process executing the algorithm for *op*. Two operations *overlap* (in time) if their intervals overlap.

The *conflict graph* of a configuration C, is an undirected graph, in which vertices represent data items and edges represent operations; it captures the distance between operations overlapping in time. If C is a configuration during the execution interval of an operation *op*, and *op* accesses the data items l_i and l_j, the graph includes an

[1] Afek et al. [1] state $O(\log^* n)$-local complexities, treating k as a constant.

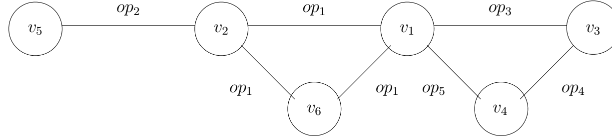

Fig. 1: The conflict graph of five overlapping operations: $op_1 = 3\mathsf{RMW}(v_1, v_2, v_6)$, $op_2 = 2\mathsf{RMW}(v_2, v_5)$, $op_3 = 2\mathsf{RMW}(v_1, v_3)$, $op_4 = 2\mathsf{RMW}(v_3, v_4)$, $op_5 = 2\mathsf{RMW}(v_1, v_4)$.

edge, labeled *op*, between the vertices v_i and v_j. (See Fig. 1.) The conflict graph of an execution interval α is the union of the conflict graphs of all configurations C in α.

The *distance* between two operations, *op* and *op*$'$, in a conflict graph, is the length of the shortest path between an item that is accessed by *op* and an item that is accessed by *op*$'$. In particular, if the operations access a common item, then the distance between *op* and *op*$'$ is zero; the distance is ∞, if there is no such path. The *d-neighborhood of an operation op* contains all the operations within distance $\leq d$ from *op*.

We use the following variant on the failure locality definition [6]:

Definition 1. *The* failure locality *of an algorithm is d if some operation in the d-neighborhood of an operation op completes after a finite number of steps by the process that invokes op, unless some operation in the d-neighborhood of op fails.*[2]

The next definition is the nonblocking analogue of failure locality; it grantees progress within every neighborhood of a small diameter, even when there are failures. This property is stronger than requiring the implementation to be *nonblocking* [9].

Definition 2. *An algorithm is d-*local nonblocking *if some operation in the d-neighborhood of an operation op completes after a finite number of steps by the process that invokes op.*

3 A Blocking Algorithm with $3k$ Failure Locality

In this section, we present a general scheme for implementing $k\mathsf{RMW}$ with bounded locality; several methods of the scheme are then implemented in a way that yields a *blocking* implementation with $3k$ failure locality. In the next section, these methods are implemented in a way that yields a local nonblocking implementation.

An operation first acquires *locks* on its data items, one item after the other, then applies its changes atomically, and finally, release the locks. A contentious situation occurs when an operation *op* is blocked since one of its items is locked by another, *blocking* operation *op*$'$. Our algorithm uses the number of data items that are already locked to decide whether to *wait* for or *reset op*$'$, i.e., release all the locks acquired by *op*$'$, and seize the lock on the required item. That is, *op* waits for *op*$'$ only if *op*$'$ is more advanced in acquiring its locks, i.e., *op*$'$ has locked more items. Otherwise, if *op*$'$ has

[2] Choy and Singh [6] require an operation to complete if no operation fails in its d-neighborhood, while we only guarantee that *some* operation in the neighborhood completes.

locked fewer items than *op*, *op* resets op'. Resetting another operation is synchronized through an *operation object* that can be acquired by operations; an operation resets another operation only after acquiring ownership on its object.

The crux of the algorithm is in handling the *symmetric* case, when *op* and op' have locked the same number of items. In this case, the algorithms breaks the symmetry by applying DCAS to atomically acquire ownership of the two operation objects (*op* and op'); the operation that acquires ownership, resets its contender. This breaks apart long hold-and-wait chains that would deteriorate the locality as well as hold-and-wait cycles that can cause a deadlock.

Detailed description. Shared memory locations are grouped in contiguous blocks, called *item objects*, which are accessed by the processes. Each item object contains a *data* attribute and a *lock* attribute. For each operation, we maintain an *operation object* containing a *dataset*, referencing the set of items the operation has to modify, a *count keeper*, which is a tuple of a *counter* holding the number of items locked so far (initially 0), and a *lock* referencing the *owner* of the operation object ($\perp$ if the object is released). The object also contains a self pointer, *initiator*.

The psudocode for the general scheme appears in Algorithm 1, while the methods for the blocking implementation appear in Algorithm 2. An operation acquires the lock on its operation object (line 3) before proceeding to acquire the locks on its data items (lines 8-9). When the operation succeeds in locking an additional item (line 10) it increases the counter (line 11); when all the items are locked, i.e., the counter is equal to the number of data items, the operation can apply its changes (line 15), and release the locks on the data items (line 16). When *op* discovers that it is blocked by another operation op' (line 12), it calls handleContention, which compares the counters of *op* and op'. If the counter of *op* is higher than (line 22) or equal to (line 27) the counter of op', *op* tries to reset op' (lines 24, 29) so as to seize the lock on the item. For this purpose, *op* needs to hold the locks on the operation objects of both *op* and op'. When the counter of *op* is higher than the counter of op', *op* keeps the lock on its operation object, and tries to acquire the lock on the operation object of op' (line 23), using CAS suffices in this case (line 5 in Algorithm 2). When the counters are equal, *op* releases the lock on its operation object (line 26) and tries to lock atomically both operation objects (line 28) by applying DCAS (line 8 in Algorithm 2). If the counter of *op* is lower (line 31), then *op* releases the lock on its operation object (line 26) and tries again.

Outline of correctness proof: Locks on items are acquired and released and counters are changed either in the locking items loop (lines 9, 11), or during a reset (lines 36, 38, 45). In both cases, locks on data items and counters are modified only after the initiator acquires the lock on its operation object (lines 3, 23, 28). Moreover, the operation holds the lock on its operation object when it has locked all its items and cannot be reset. Thus, changes are applied (line 15) in isolation, implying that the algorithm is linearizable.

Several types of blocking and delay chains might be created during an execution of the algorithm. Some of these chains are created when an operation fails and causes other operations to wait. It is intuitively clear why the length of these chains is in $O(k)$.

More intricate delay chains are created when operations reset other operations. For example, assume an operation op_1 resets another operation op_2, then, a third operation

Algorithm 1 Multi-location read-modify-write: general scheme

```
 1: run() {
 2:     while (c ← READ(initiator.countKeeper.counter)) < size do
 3:         if initiator.lockOperation(initiator, c) then              // lock this operation object
 4:             initiator.execute()
 5: }
 6: execute() {
 7:     while (c ← READ(initiator.countKeeper.counter)) < size do      // more items to lock
 8:         item ← READ(initiator.dataset[c])
 9:         CAS(item.lock, ⊥, initiator)                               // acquire lock on item
10:         if READ(item.lock) = initiator then
11:             CAS(initiator.countKeeper, ⟨c,initiator⟩, ⟨c+1,initiator⟩)    // increase counter
12:         else                                          // initiator is not the owner of the item
13:             initiator.handleContention(item)
14:             return
15:     write modified values to the memory locations
16:     initiator.unlockDataset()
17: }
18: handleContention(Item item) {
19:     if (conflict ← READ(item.lock)) = ⊥ then return                 // no conflict on item
20:     ⟨ic,iowner⟩ ← READ(initiator.countKeeper)
21:     ⟨cc,cowner⟩ ← READ(conflict.countKeeper)
22:     if ic > cc then                       // conflict with an operation with a lower counter
23:         if initiator.lockOperation(conflict, cc) then
24:             initiator.reset(conflict, item)
25:         return
26:     CAS(initiator.countKeeper, ⟨ic,initiator⟩, ⟨ic,⊥⟩)          // release this operation object
27:     if ic = cc then                       // conflict with an operation with an equal counter
28:         if initiator.lockTwoOperations(initiator, ic, conflict, cc) then
29:             initiator.reset(conflict, item)
30:         return
31:     if ic < cc then                       // conflict with an operation with a higher counter
32:         initiator.handleHigherConflict(conflict)
33: }
34: reset(Operation conflict, Item item) {
35:     c ← READ(initiator.countKeeper.counter)
36:     CAS(item.lock, conflict, initiator)                            // seize lock on item
37:     if READ(item.lock) = initiator then
38:         CAS(initiator.countKeeper, ⟨c,initiator⟩, ⟨c+1,initiator⟩)        // increase counter
39:     conflict.unlockDataset()
40: }
41: unlockDataset() {
42:     ⟨c,owner⟩ ← READ(initiator.countKeeper)
43:     for i = 0 to c do
44:         item ← READ(initiator.dataset[i])
45:         CAS(item.lock, initiator, ⊥)                               // release lock on item
46:     CAS(initiator.countKeeper, ⟨c,owner⟩, ⟨0,⊥⟩)  // reset counter, release operation object
47: }
```

Algorithm 2 Multi-location read-modify-write: methods for the blocking algorithm

```
1: handleHigherConflict(Operation conflict) {
2:     nop                                                   // (blocking) busy wait
3: }
4: boolean lockOperation(Operation op, int c) {
5:     return CAS(op.countKeeper, ⟨c,⊥⟩, ⟨c,initiator⟩)
6: }
7: boolean lockTwoOperations(Operation iop, int ic, Operation cop, int cc) {
8:     return DCAS(iop.countKeeper,cop.countKeeper,⟨ic,⊥⟩,⟨cc,⊥⟩,⟨ic,initiator⟩,⟨cc,initiator⟩)
9: }
```

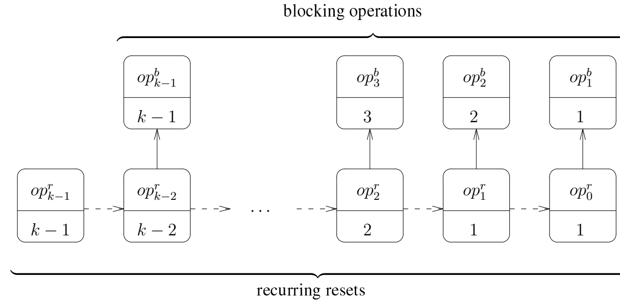

Fig. 2: A recurring resets scenario. The number below an operation indicates its counter; solid arrows indicate blocking and dashed arrows indicate reset.

resets op_1. At some later time, op_2 and op_1 can reacquire their locks, and the same scenario may happen over and over again. It may even seem as if a livelock can happen due to a cycle of resetting operations.

Since this behavior is the least intuitive, we start with the most delicate part of the proof, bounding the number of times an operation can be reset, before some operation completes in its k-neighborhood.

Fig. 2 presents an example where the superscript r denotes an operation that applies (and suffers from) recurring resets, and b denotes blocking operation that cause the operations in the recurring reset chain to release the locks on their operation objects. In the example, op^r_0 is blocked by op^b_1 with equal counter, 1, holding the lock on the item t_1. So, op^r_0 releases the lock on its operation object (in order to try and reset op^b_1 and seize the lock on t_1). op^r_1 is blocked by op^r_0 holding the lock on the item s_0. op^r_1 resets op^r_0, seizes the lock on s_0, and increases its counter to 2. At this point, op^r_1 is blocked by op^b_2 with equal counter, 2. In a similar way, op^r_2 resets op^r_1 (holding the lock on the item s_1) after it releases the lock on its operation object, and op^r_3 resets op^r_2. Then op^r_0 and op^r_1 are able to reacquire the locks on s_0 and s_1 respectively. This scenario is repeated with longer chains of resets each time. Inspecting the example reveals that after $k/2$ resets of op^r_0, op^r_{k-1} at distance $k-2$ from op^r_0 locks all its data items, and it can complete.

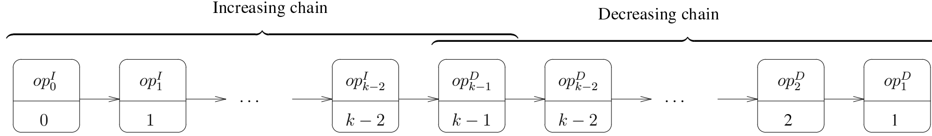

Fig. 3: The operations op_j^I create an increasing chain; each op_j^I operation is blocked by the operation op_{j+1}^I holding the lock on the data item t_{i+1}. The operations op_i^D create a decreasing chain; each op_i^D operation is blocked by the operation op_{i-1}^D holding the lock on its operation object.

Let $c(op)$ be the counter of op in a given configuration. For an operation op, k is the maximal number such that some operation that overlaps op in time, is within the $3k$-neighborhood of op and accesses k data items; n_k is the number of operations in the k-neighborhood of op.

Since an item is seized during a reset, we can prove a lemma stating that after each time an operation is reset, some operation makes progress. This serves as the base case for another lemma, proving that some set of operations make progress after each reset. This is used to prove the following lemma, arguing that after an operation is reset a bounded number of times, some operation in its $(k-1)$-neighborhood completes.

Lemma 1. *If the operation object of op is released (and re-acquired) $(n_k)^2k$ times, then some operation completes in the $(k-1)$-neighborhood of op.*

We next describe how to bound the blocking chains created due to failures.

Consider an operation op that fails while holding the lock on its operation object and a data item. Another operation op' needing this item cannot complete without resetting op. If $c(op') > c(op)$, op' continues to hold the lock on its operation object, thus it may block a third operation with higher counter that cannot reset op', and so on. Fig. 3 shows such a chain, called a *decreasing chain*.

A decreasing chain consist of *stuck* operations, either initiated by a failed process or unable to increase their counter beyond some value. An operation op considers another operation op' *stuck at* m in an execution interval α, if in any configuration C during α in which op needs to reset op' (since op' blocks op), $c(op') \leq m$, and some operation (possibly op') has the lock on the operation object of op'.

Consider, for example, the operation op_2^D in Fig. 3 that needs to acquire the lock on a data item s_1, is blocked by another operation op_1^D with a lower counter that has the lock on its operation object. op_1^D does not complete either because it does not take steps or because it is repeatedly being reset by other operations, and always before op_2^D has a chance to acquire the lock on s_1, op_1^D reacquires the lock on its operation object and on s_1. Thus op_2^D considers op_1^D stuck at 1. For every i, $1 < i < k-1$, the operation op_{i+1}^D in Fig. 3 considers the operation op_i^D stuck at i, since op_i^D holds the lock on its operation object while blocking op_{i+1}^D, leading to the decreasing chain.

Another blocking scenario is when an operation op that needs to acquire a lock on a data item is blocked by another operation whose counter is higher than $c(op)$. Moreover, op may block a third operation with a lower counter, and so on, creating an *increasing chain*, also depicted in Fig. 3. Since each operation in an increasing chain has counter

that is strictly higher than the counter of the operation that it blocks, the length of the chain can easily be bounded by k.

Decreasing and increasing chains, together with recurring resets may create a longer delay chain. We show that these are the only ways to combine delay chains. In an *increasing-decreasing chain* of operations, every operation *op* is blocked either by the next operation in the chain op' with $c(op') > c(op)$ or by a decreasing chain that starts at op' with $c(op') \leq c(op)$. The next lemma bounds the length of increasing-decreasing chains and yields the bound on the failure locality.

Lemma 2. *Let m, $0 \leq m \leq k$, be the counter of an operation op, and assume no operation on an increasing-decreasing chain idc from op fails, and all the operations that reset an operation on idc complete the reset. Then some operation in the $(3k - m - 2)$-neighborhood of op completes after a finite number of steps by op.*

Theorem 1. *Algorithm 1 with the methods of Algorithm 2 has $3k$ failure locality.*

4 The $3k$-Local Nonblocking Algorithm

A $3k$-local nonblocking implementation is obtained by incorporating a recursive *helping* mechanism, as in other multi-word synchronization algorithms [5,8,14,15]. When a process p, executing an operation *op*, is blocked by another operation op' with a higher counter, *p helps op'* to complete and release the item, instead of waiting for the process p' executing op' to do so by itself (which may never happen due to the failure of p'). Helping means that p executes the protocol of op' via the helping method (we say that *op helps op'*). Helping is recursive, thus if while executing op', p discovers that op' is blocked by a third operation op'', then p recursively helps op''. Note that *op* still resets op' if the counter of *op* is equal or higher than the counter of op'. Special care is needed since *op* can also be blocked while trying to lock an operation object; in this case also, *op* helps the blocking operation.

The methods for the nonblocking algorithm appear in Algorithm 3. The first difference is that an operation *op*, blocked by another operation op' with higher counter (handleHigherConflict), helps op' to complete and release its data items (line 2).

While trying to acquire the lock on an operation object (lockOperation or lockTwoOperations), an operation *op* may succeed (line 16) or be blocked by another operation op'. If op' is the initiator of this operation (line 18), then *op* helps op' to complete, and release the operation object (line 19); otherwise (line 20), op' locked the operation object in order to reset its initiator, so *op* only helps op' to complete the reset (line 23).

In the helping scheme, several *executing processes* execute an operation. Helping is synchronized with CAS primitives to ensure that only one executing process performs each step of the operation, and the others have no effect.

The execution interval of an operation in the general scheme is divided into disjoint *rounds*, each starting after the lock on its object is released. If an executing process p discovers (while executing some operation *op*) that it is blocked by op' in its r-th round, it helps op' only in the context of this round. If the round number of op' changes, then op' released its operation lock, and the set of items locked by op' might have

Algorithm 3 Multi-location read-modify-write: methods for the nonblocking algorithm

```
1:  handleHigherConflict(Operation conflict) {
2:      conflict.execute()                                   // help execute the operation
3:  }
4:  boolean lockOperation(Operation op, int c) {
5:      CAS(op.countKeeper, ⟨c,⊥⟩, ⟨c,initiator⟩)
6:      return initiator.verifyLock(op)
7:  }
8:  boolean lockTwoOperations(Operation iop, int ic, Operation cop, int cc) {
9:      DCAS(iop.countKeeper, cop.countKeeper, ⟨ic,⊥⟩, ⟨cc,⊥⟩, ⟨ic,initiator⟩, ⟨cc,initiator⟩)
10:     if initiator.verifyLock(iop) then
11:         return initiator.verifyLock(cop)
12:     return FALSE
13: }
14: boolean verifyLock(Operation op) {
15:     ⟨c,owner⟩ ← READ(op.countKeeper)
16:     if owner = initiator then
17:         return TRUE                         // succeeded locking the operation object
18:     if owner = op.initiator then  // the initiator of the operation owns the operation object
19:         op.execute()                                     // help execute the operation
20:     else                                 // a third operation owns the operation object
21:         oc ← READ(owner.countKeeper.counter)
22:         item ← READ(owner.dataset[oc])
23:         owner.reset(op,item)                              // help reset the operation
24:     return FALSE                               // failed locking the operation object
25: }
```

changed. As in the blocking algorithm, we omit details such as round numbers and ABA-prevention tags from the code, for clarity.

By showing that the executing processes are correctly synchronized by the round numbers and ABA-prevention tags, we can prove that the algorithm is linearizable. The locality properties of the algorithm are proved by reduction to the failure locality of the blocking algorithm. The next lemma shows that if a process takes many steps, it will eventually get to help any process that could be blocking it to make progress, thereby alleviating the effect of their failure.

Lemma 3. *If an executing process of an operation op takes an infinite number of steps, then an executing process of each of the operations on any increasing-decreasing chain idc from op takes an infinite number of steps and all the operations that reset an operation on idc complete the reset.*

Lemmas 2 and 3 imply that the algorithm is local nonblocking; similar ideas show that the algorithm has $3k$-local step complexity and $4k + 1$-local contention.

Theorem 2. *Algorithm 1 with the methods of Algorithm 3, is $3k$-local nonblocking.*

5 Discussion

We have presented a kRMW algorithm with improved throughput even when there is contention. Like Afek et al. [1], we can make this algorithm be wait-free by applying a known technique [2] while maintaining the locality properties of the algorithm.

Our algorithm has $O(k)$-locality properties, when using DCAS, and $O(k + \log^* n)$-locality properties, when using only CAS. It is theoretically interesting to obtain locality properties that are independent of n, without using DCAS. Even more intriguing is to investigate whether $O(k)$ is the best locality that can be achieved, even with DCAS.

Our algorithmic ideas can serve as the basis for *dynamic* STM. This is because our algorithm needs to know the identity of a data item only as it is about to lock it and can be adapted to work when data items are given one-by-one. Realizing a full-fledged STM requires to address many additional issues, e.g., handling read-only data, and optimizing the common case, which are outside the scope of this paper.

Acknowledgements. We thank Rachid Guerraoui, David Hay, Danny Hendler, Alex Kogan and Alex Shraer 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. J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *PODC '95*, pages 184–193.
3. H. Attiya and E. Dagan. Improved implementations of binary universal operations. *J. ACM*, 48(5):1013–1037, 2001.
4. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley& Sons, second edition, 2004.
5. G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA 1993*, pages 261–270.
6. M. Choy and A. K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Trans. Program. Lang. Syst.*, 17(3):535–559, 1995.
7. P. H. Ha, P. Tsigas, M. Wattenhofer, and R. Wattenhofer. Efficient multi-word locking using randomization. In *PODC 2005*, pages 249–257.
8. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC 2002*, pages 265–279.
9. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
10. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*, pages 289–300.
11. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
12. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC 1994*, pages 151–160.
13. R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS 2002*, pages 5–17.
14. N. Shavit and D. Touitou. Software transactional memory. *Dist. Comp.*, 10(2):99–116, 1997.
15. J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *PODS 1992*, pages 212–222.