

Space- and Time-adaptive Nonblocking Algorithms

Maurice Herlihy^a Victor Luchangco^b Mark Moir^b

^a *Computer Science Department, Brown University, Providence, RI 02912, USA*

^b *Sun Microsystems Laboratories, Burlington, MA 01803, USA*

Abstract

We explore techniques for designing nonblocking algorithms that do not require advance knowledge of the number of threads that participate, whose time complexity and space consumption both adapt to various measures (rather than being based on predefined worst-case scenarios), and that can continue to reclaim memory even after thread failures. The techniques we introduce can be implemented using widely available hardware synchronization primitives. We present our techniques in the context of solutions to the well-known *Collect* problem. We also explain how our techniques can be exploited to achieve other results with similar properties, such as long-lived renaming and dynamic memory management for nonblocking data structures. In addition to the algorithmic techniques we introduce, we also clarify and generalize previous properties used to characterize measures of an algorithm's "adaptivity".

1 Introduction

Lock-free and wait-free (collectively nonblocking) implementations of shared data structures are designed to overcome numerous problems associated with the common use of mutual exclusion locks, including deadlock, convoying, performance bottlenecks, and, in real-time systems, priority inversion. A *wait-free* implementation guarantees that each operation on the data structure completes after a finite number of its own steps, regardless of the timing behaviour of threads executing other operations. A *lock-free* implementation guarantees that after a finite number of steps of an operation, *some* operation completes. Both definitions preclude the use of locks for synchronization.

Significant efforts have been made over the last two decades towards practical nonblocking synchronization, and much progress has been made. However, most nonblocking algorithms require *a priori* knowledge of the number N of threads that will *potentially* participate, and they behave incorrectly if N is underestimated. We call such algorithms *population-aware*; algorithms

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

that do not require such knowledge are *population-oblivious*. Because many population-aware algorithms have space consumption and/or time complexity that depend on N , overly conservative estimates of N result in wasted time and space. Prior research efforts (e.g., [1,2,6,16]) have addressed the time complexity part of this problem by designing algorithms whose time complexity “adapts” to the number of threads that *actually* participate. Most of these research efforts have been based on the assumption that only read and write operations are available, which is not true in modern shared-memory multiprocessors. This restriction has led to algorithms that are ingenious and beautiful—but ultimately impractical. It has also prevented the work from addressing either the space overhead problem or the need for a known bound on the number of threads that *potentially* participate. Nonetheless, this work has been important in drawing attention to these issues, as well as in beginning to formulate appropriate measures for characterizing algorithms that overcome some of these problems. Other results (e.g., [15]) do not require knowledge of N , and can allocate space dynamically as required, but cannot reclaim space after it has been used. To our knowledge, the only previous algorithm that does not have any of these shortcomings can be prevented from future memory reclamation by a single thread failure [18].

In this paper, we present simple techniques that can be used to address some or all of these problems in a variety of contexts. They are based on the compare-and-swap (CAS) instruction, which is widely available in modern shared-memory multiprocessors. We present our techniques in the context of solutions to the well-known *Collect* problem [6,17], which is a building block in solutions to various problems in concurrent computing. We also explain how the same techniques can be applied to achieve solutions to other problems that overcome some of the above-mentioned shortcomings with previous solutions.

Because the algorithms presented here allocate and free memory dynamically, they are not truly nonblocking unless they are used in conjunction with a nonblocking memory allocator. Recently, Dice and Garthwaite [9] proposed a new memory allocator. Their goal was to make the allocator “multiprocessor-aware” in order to improve locality and reduce synchronization overhead. As a result, their allocator resorts to locking only very occasionally, and therefore scales very well. While their allocator does still use locks, variations on straightforward and well-known nonblocking techniques could be applied to make it completely nonblocking. We therefore do not address this problem further.

1.1 Our contributions

We consider the well-known *Collect* problem [6,17], and present population-oblivious, nonblocking solutions that are adaptive in both time and space. In the *Collect* problem, threads can store values, and can “collect” a set of recently-stored values. The *Collect* problem is defined precisely in the next

section.

We present several Collect solutions with different adaptivity properties. We also explain how the same techniques can be used to achieve population-oblivious, time- and space-adaptive solutions to several other problems for which such solutions did not previously exist. These include renaming [1,5,16] and memory management for dynamic-sized nonblocking data structures. In particular, these techniques can be used to make our nonblocking memory management scheme [11] population-oblivious and time- and space-adaptive. Thus, using the results of Herlihy, et al. [10], we can achieve a version of Michael and Scott’s lock-free FIFO queue that is population-oblivious and time- and space-adaptive. To our knowledge, this is the first such implementation of any shared data structure that cannot be prevented from subsequent memory reclamation by thread failures.

Our first Collect solution is extremely simple, but its space consumption depends on “historical” measures (defined in the next section), which might be acceptable in some applications, but not in others. To overcome this shortcoming, we present another, somewhat more complicated implementation that removes the dependence on the historical measures, but can be prevented for a long time from reclaiming space by continuous series of overlapping Collect operations. We also explain how to modify it to overcome this problem. All of these solutions are lock-free and are based on compare-and-swap (CAS), and can therefore be implemented on a wide variety of modern shared-memory multiprocessors. We also explain how our algorithms are wait-free under various assumptions. In some cases, the assumptions include the existence of exotic, but nonetheless implementable, hardware instructions.

2 Adaptivity Properties

There are various properties that can be used to characterize algorithms with respect to their time complexity and space consumption. Below, we describe related previous work that has led to some of these properties, and also propose a new structure for expressing adaptivity measures; this is intended to clarify and generalize previous definitions. We believe that this important for understanding and characterizing the adaptivity properties of algorithms. On the other hand, the rest of the paper is independent of this discussion, so readers who primarily interested in algorithms are encouraged to skip to Section 3, and read the remainder of this section later in order to follow our analysis of the adaptivity properties of our algorithms.

2.1 Time Adaptivity Properties and Related Work

In some applications, the number of threads created depends on factors such as the machine on which the application is run, application input, or asynchronous and unpredictable events. Furthermore, it may be difficult or im-

possible to provide tight upper bounds on the maximum actual contention for a particular shared data structure. Thus, implementations whose time complexity and space consumption adapt to the *actual* conditions in each execution—rather than being determined by *a priori* bounds on *worst-case* conditions—are preferable. Below we discuss what we mean by “actual conditions” more precisely.

Previous work in this area has focused on implementations whose time complexity adapts to contention. In this context, various definitions of “contention” and “adapt” are useful for different purposes. For example, contention during a particular interval of time might mean the total number of distinct threads that are active during that interval, or it might mean the maximum number of distinct threads that are *simultaneously* active at any point in time during that interval. These two definitions were called *interval contention* and *point contention*, respectively, by Afek, et al. [1]. We think these names are unfortunate because both are defined with respect to an interval of time; we would prefer to call them *cumulative* and *concurrent* contention, respectively.

In general, adaptivity properties need to be expressed in terms of a variety of different measures; as demonstrated by several examples in this paper, merely considering contention is not always sufficient. Furthermore, adaptivity properties are often expressed as functions of certain measures over certain intervals of time. The interval considered might be the execution interval of an operation (*operation*), or the interval starting from the beginning of the execution history up until the current time (*historical*), or some other interval (we present other choices later in the paper). The choice of interval affects what it means for an implementation to adapt. For example, if an implementation guarantees that the time complexity of a particular operation is bounded by a function of the maximum number of operations simultaneously executing at any point during the execution of that operation, and is independent of the number of threads that were active at any point before the operation began execution, then we would say that the operation implementation is *time-adaptive to operation-concurrent contention*. Alternatively, if an implementation guarantees that the operation’s time complexity is bounded by a function of the maximum number of threads executing concurrently at any point in the past, we would say that the operation implementation is *time-adaptive to historical concurrent contention*.

There has been considerable research in recent years on implementations that are time-adaptive for various definitions. Some pioneering efforts in this direction are the universal constructions of Afek, Dauber, and Touitou [2], one-shot and long-lived renaming algorithms [1,4,16], and Collect implementations [6]. This work has resulted in algorithms that are time-adaptive under a variety of definitions. However, all of them are population-aware because space must be preallocated for the possibility that all N threads access the implementation concurrently. Space consumption for each of the read/write-

based adaptive long-lived renaming and Collect algorithms is at least cubic in N , and in some cases is exponential in N or even unbounded. This space must be allocated in advance, so it is used even if actual contention is always very low (and results that use unbounded space are unimplementable). In the next section we discuss efforts to overcome this shortcoming.

2.2 *Space Adaptivity Properties and Related Work*

In this paper we present the first solutions to all of the problems mentioned above in which space consumption adapts to actual conditions, rather than worst-case conditions. A side-effect of this effort is that we have achieved algorithms that are population-oblivious, a significant advantage.

Designing implementations whose space consumption adapts to actual conditions can introduce a significant additional challenge, depending on the chosen measure for “actual conditions”. In particular, if this measure can decrease over time, the space consumption of the implementation must also decrease, i.e., memory must be freed. As observed in recent work on memory management for nonblocking implementations of shared data structures [11,14], freeing memory in nonblocking implementations is challenging because it is difficult to ensure that another thread will not access a memory block after it has been freed. This explains why previous work has not addressed the issue of space consumption and why our work depends on strong synchronization primitives, while progress was made on time-adaptive implementations in work that used only reads and writes. Below we discuss previous nonblocking algorithms that are space adaptive, and also discuss various measures to which space consumption can be required to adapt.

Treiber [18] presents a population-oblivious, lock-free implementation of a linked-list-based set data structure that can reclaim memory after use. However, the space consumption of this implementation can be caused to grow without bound, independent of historical measures, by a single failure or by continuous access.

In recent years, our group has proposed a variety of population-oblivious, lock-free implementations of dynamic-sized double-ended queue data structures (dequeues) [3,7,13] that have space consumption that is independent of historical measures. These implementations assume the presence of a garbage collector, which significantly simplifies design, but our group has also demonstrated a method for removing this reliance [8]. However, this work depends on a double compare-and-swap (DCAS) operation, which is not widely supported in hardware.

Michael and Scott [15] present a population-oblivious, lock-free FIFO queue implementation that is space-adaptive to historical maximum queue size. To make this more precise, let us define *queue size at time t* as the total number of enqueue operations that have ever been invoked before time t minus the number of dequeue operations that have returned a value other than “empty”

before time t , and let us define the *historical maximum queue size at time t* as the maximum queue size at any time $t' < t$. Then the space consumption of Michael and Scott’s queue implementation at time t is bounded by a function of historical maximum queue size. Although this algorithm eliminates the need to preallocate space for the maximum future queue size (and therefore to estimate a bound on that maximum), it has the disadvantage that if the queue grows large and subsequently shrinks, the space no longer required cannot be reclaimed. This is because their implementation stores dequeued nodes in a pool (otherwise known as a freelist); nodes in the pool can be reused later by subsequent enqueue operations, but can never be freed because we can never be sure that they won’t subsequently be accessed.

We recently presented a technique for lock-free memory management [11]. We also demonstrated how to use this technique to modify Michael and Scott’s implementation so that it can free nodes from its pool [10]. Maged Michael also recently proposed a similar technique to achieve a similar result [14]. (See our memory management paper [11] for a comparison.) We refer the reader to those papers for details of the techniques; here we discuss only the aspects of those approaches that are relevant to our work here. In our approach, “guards” are “posted” on values to prevent them from being freed prematurely. In Michael’s approach, “hazard pointers” are used for the same purpose (we use “guards” to refer to both our guards and Michael’s hazard pointers where the distinction is not important). In both papers, the guards are implemented as preallocated arrays. Because each thread must use some number of guards, this requires both approaches to know in advance the maximum number of guards. In fact, in Michael’s approach, the number of threads and the number of guards per thread must be known in advance, whereas in ours only the maximum total number of guards needs to be known because our approach can allocate guards dynamically. In practice, both approaches require the number of threads to be known in advance. However, we claimed that paper that it is straightforward to remove this restriction from our approach, thereby making it population-oblivious and space-adaptive; we elaborate in this paper.

3 Dynamic Collect Problem

Below we formally define the Collect problem. Our definition follows the spirit of the one by Attiya, Fouren and Gafne [6], but is modified slightly to make it dynamic. Roughly speaking, in the formulation of Attiya, et al., each thread owns a single location in which it can *store* a value, and threads can *collect* a set of up-to-date values from all threads that have stored values. Our formulation generalizes this to allow threads to have multiple values stored in multiple locations at a time, and also to release those locations when they are no longer required so that the memory used for them can be reclaimed (or subsequently reused by another thread). We call our more general problem the *Dynamic Collect* problem, but for brevity we generally call it the Collect

problem (we do not discuss the original Collect problem further in this paper, so there should be no confusion).

A solution to the Collect problem is an implementation of a Collect object, which is defined as follows using two data types, *address* and *value*. A Collect object supports the following operations:

- *Register()*: returns an address
- *Store(a, v)*: stores value v at address a
- *Deregister(a)*: deregisters address a
- *Collect()*: returns a set of (address,value) pairs

We say that an address a is *registered* to a thread p when it is returned by an invocation of *Register* by p , and that it is *deregistered* when *Deregister(a)* is invoked. A thread may invoke *Store* and *Deregister* only with an address that has previously been registered to it, and which it has not since deregistered. Any thread may invoke *Collect* at any time. A thread may invoke *Register* multiple times without deregistering addresses previously registered to it.

Register returns an address that has either never been registered, or has been deregistered since the previous time it was registered.

Following standard definitions, we say that if the first event of an operation $op0$ occurs after the last event of another operation $op1$, then $op0$ *follows* $op1$ and $op1$ *precedes* $op0$.

A *Collect* operation cop returns a set S of values such that the following conditions hold:

- For an address a , if there does not exist a value w such that $(a, w) \in S$, then either there does not exist a *Store(a, v)* operation sop for any v such that sop precedes cop , or there exists a *Deregister(a)* operation dop that does not follow cop such that there is no *Store(a, v)* operation $sop2$ for any value v such that dop precedes $sop2$ and $sop2$ precedes cop .
- For an address a and a value v , if $(a, v) \in S$, then there exists a *Store(a, v)* operation sop that does not follow cop such that there does not exist an operation $op1$ that is either a *Store(a, w)* operation for some $w \neq v$ or a *Deregister(a)* operation, such that sop precedes $op1$ and $op1$ precedes cop .

The above definition is weak in that it does not require *Store* operations to be atomic. For example, it does not preclude the possibility that a *Collect* operation that “sees” the value of a concurrent *Store* operation precedes another *Collect* operation that does not see it. It turns out that our algorithms do implement atomic *Store* operations, as do previous implementations such as those presented by Attiya, et al. [6]. To be more precise, each *Store* operation in these implementations can be viewed as if it were a single event that occurs sometime between its first and last events (inclusive).

4 Algorithms

In this section, we present two Collect algorithms that are population-oblivious and are adaptive in both time and space. To more precisely state the adaptivity properties, we define the *activity level at time t* to be the sum of the number of addresses registered at time t and the number of threads either registering or deregistering an address at time t . Our first algorithm is very simple, and supports constant-time *Store* and *Deregister* operations and a *Register* operation whose time complexity adapts to the maximum activity level experienced during that operation. However, in this algorithm, both the time complexity of the *Collect* operation and space consumption adapt only to the historical maximum activity level; that is, this algorithm cannot free memory that is no longer required. Our second algorithm also supports a constant-time *Store* operation and a *Register* operation whose time complexity adapts to the maximum activity level experienced during that operation. In this algorithm, the time complexity of the *Deregister* operation is bounded by the maximum activity level during the *Register* operation that registered the address being deregistered. This algorithm can free memory, even after thread failures, but can be prevented from freeing memory by a continuous series of overlapping *Collect* operations; also the time complexity of the *Collect* operation is proportional to the memory consumption, so these concurrent *Collect* operations also keep their own time complexity high in this scenario. Finally, we explain how to modify the second algorithm so that it does not have this problem. We state the properties of the modified algorithm more precisely later.

Each of our algorithms maintains a linked list of nodes that store values. When the activity level exceeds the number of nodes in the list, new nodes may be allocated and appended to the end of the linked list. Nodes representing addresses that have been deregistered may be reused and returned in response to later registrations. We present and discuss each of these algorithms in detail in the rest of this section.

4.1 Algorithm 1: Space Adaptivity to Historical Activity Level

The data types used in Algorithm 1 are shown in Figure 1, and C-like pseudocode for the algorithm appears in Figure 2.¹ The code assumes there is one node that is initially allocated and set to `(false, (? , false), null)`, where `?` indicates that any value is acceptable. The address of this node—call it `Head`—is known to all threads.

¹ For simplicity, we present our algorithms for a machine that provides a sequentially consistent [12] shared memory. For machines that provide weaker memory models, additional memory barriers must be inserted to ensure correctness; this is generally straightforward.

```

valtype = record                                // stored together in atomically storable location
    valuetype value;                            // valuetype is whatever type Store stores
    bool    valid;
end;

nodetype = record
    bool    owned;
    valtype val;
    nodetype *next;
end

```

Fig. 1. Data types used by Algorithm 1.

```

nodetype * Register() {
    nodetype *nd;
    nodetype *newnode = new nodetype;        // Allocate new node just in case
    newnode->owned = true;                    // It's mine, all mine...
    newnode->val = (?,false);                 // Not a stored value; ? = don't care
    newnode->next = null;                     // Terminate node

    nd = Head;                               // Start with first node
    while (true) {
        if (CAS(&nd->owned,false,true)) {    // Try to claim node
            delete newnode;                 // Didn't use newnode
            return nd;                       // Success; node claimed
        }
        if (nd->next == null)                // No more nodes
            if (CAS(&nd->next,null,newnode)) // Try to install one
                return newnode;             // Success; return node
        nd = nd->next;                       // If previous CAS failed, someone else
    }                                        // added a new node; go to it
}

Store(nodetype *nd, valuetype v) {
    nd->val = (v,true);                       // just store value with valid indicator
}

Deregister(nodetype *nd) {
    nd->val = (?,false);                      // so value won't get returned later
    nd->owned = false;                       // give up ownership
}

valueset Collect() {
    valueset S = emptyset;                   // no values so far
    nodetype *nd = Head;                    // start at first node
    while (nd) {                             // search through all nodes
        v = nd->val;                          // get value
        if (v.valid)                          // if it's a valid value,
            S = S union {(nd,v.value)};      // add it to the set
        nd = nd->next;                       // get next (if any)
    }
    return S;                                // return set of values collected
}

```

Fig. 2. Algorithm 1.

4.2 Algorithm 2

Each node has a flag `owned` that indicates whether it is registered to some thread. To register, a thread p walks down the list attempting to claim an unowned node (by changing its `owned` flag from `false` to `true`). If it successfully claims such a node, then it returns that node as the result of the registration. If p reaches the end of the list without successfully claiming any node, then it attempts to add a new node (with the `owned` bit already set to `true`) to

the end of the list. If successful, p returns that new node. Otherwise, the list has been extended by some other thread, so p continues down the list, attempting to claim an unowned node. To ensure that nodes are not claimed by multiple threads, and that new nodes are added successfully to the end of a list before they are returned, an atomic compare-and-swap (CAS) operation² is used to claim and to install nodes. To prevent *Collect* operations from returning values from nodes that have not yet had a value stored in them, or that have been deregistered, the value field of each node contains a `valid` flag that is initialized to *false* before the node is linked into the list, and is reset to *false* when the node is deregistered; *Store* operations atomically set this bit to *true* when writing their values (it would be straightforward to separate these stores to allow stored values to take up an entire atomically-storable location). The *Collect* operation walks down the list recording the valid (node, value) pairs it encounters as it does. The *Deregister* operation sets the `valid` bit of the node being deregistered to *false*, as explained above, and then sets the `owner` flag to *false*, so that the node may be claimed by future *Register* operations. The *Store* and *Deregister* operations do not need to be implemented with CAS because no other thread will concurrently write the node accessed by these operations. (Recall that a thread may only execute the *Store* and *Deregister* operations on an address that it has previously registered and not since deregistered.)

The registration and deregistration part of this algorithm is a variation of a test-and-set-based long-lived renaming algorithm of Moir and Anderson [16]. We can show that at some point in the execution of a *Register* operation that returns the k th node, the activity level is at least k (see Appendix A). Because a *Register* operation always returns a node if it adds that node to the list, this property implies that a new node is added to the list only if the activity level is greater than the number of nodes in the list. However, because nodes are never removed from the list, the list does not shrink when the activity level decreases. Thus, the space consumption of this algorithm adapts to the historical maximum activity level.

The *Store* and *Deregister* operations both take constant time (and use only store instructions). The *Register* operation takes time proportional to the number of nodes it checks to see if they are claimed, which, by the property discussed above, is bounded by a function of the maximum activity level at any time during the execution of the *Register* operation. The *Collect* operation reads every node in the list, so it takes time proportional to the length of the list, which is bounded by a function of the historical maximum activity level.

Our second algorithm improves on the first one by allowing nodes to be

² A $\text{CAS}(\mathbf{a}, \mathbf{e}, \mathbf{n})$ operation accepts three parameters: an address \mathbf{a} , an expected value \mathbf{e} , and a new value \mathbf{n} . If the value currently stored at address \mathbf{a} matches the expected value \mathbf{e} , then CAS stores the new value \mathbf{n} at address \mathbf{a} , and returns *true*; we say that the CAS *succeeds* in this case. Otherwise, it returns *false* and does not modify memory; we say that the CAS *fails* in this case.

removed from the list and freed. Thus, unlike the first algorithm, the space consumption of the *Collect* object in this algorithm, and the time complexity of the *Collect* operation, can decrease when the activity level decreases. To avoid memory access faults, before a thread may free a node, it must ensure that no other thread will subsequently access that node [18]. To do this, we use “hold counts”, as explained below.

The algorithm is based on doubly linked lists. As before, a thread registers by finding and claiming a node in the list; if it reaches the end of the list without success, it extends the list with a newly allocated node; a back pointer is stored in the node before it is added to the end of the list, so the list is always doubly linked. To facilitate the reclamation of nodes, we augment each forward pointer in the linked list with a counter, which we call the *hold count*. This counter is read and updated atomically with the pointer. Specifically, when a thread executing either *Register* or *Collect* reads a pointer while walking down the list, it also increments that pointer’s associated counter. Also, when a *Collect* operation has reached the end of the list, it follows the back pointers in the list and decrements each of the counters it incremented while walking down the list. Similarly, a *Deregister* operation walks back to the beginning of the list from the node it is deregistering, decrementing each counter that the corresponding *Register* operation previously incremented. Thus, the counter in the forward pointer of a node records the sum of the number of nodes registered in the tail of the list starting from that pointer, plus the number of threads that are seeking or accessing nodes in that tail. Therefore, if a hold count goes to zero, no threads are accessing nodes in the tail of the list from that point, and no nodes in the tail are registered. Thus, the nodes in the tail of the list can be freed. In the simple algorithm we present to demonstrate this technique, we free nodes whenever possible; it would be easy to implement less aggressive policies to avoid prematurely freeing nodes if there is reason to believe they will be needed again soon.

As with Algorithm 1, if a *Register* operation returns the k th node, then at some point during its execution, the activity level is at least k . Thus, the list grows only when the activity level is greater than the number of nodes in the list during some *Register* operation, and the space consumption is bounded by the historical maximum activity level. In addition, unlike Algorithm 1, the space consumption may decrease if the activity level decreases. However, as we discuss later, this is guaranteed only if there is a point of low activity at which no *Collect* operations are executing. At the end of this section, we describe how to modify this algorithm to achieve a stronger guarantee.

The data types used by Algorithm 2 are shown in Figure 3. One node is assumed to be allocated initially to contain $(\text{null}, \text{false}, (?, \text{false}), (0, \text{null}))$. The address of this node—call it *Head*—is known to all threads.

To describe this algorithm, we use two operations, *BumpUpOrInstall* and *BumpDownOrRemove*, to atomically update a forward pointer and its associated hold count. These operations are both simple read-modify-write op-

```

ptrctr = record
    int    ctr;
    nodetype *ptr;
end

valtype = record // stored together in atomically-storable location
    valuetype value; // valuetype is whatever type Store stores
    bool    valid;
end;

nodetype = record
    nodetype *back;
    bool    owned;
    valtype val;
    ptrctr  forward;
end

```

Fig. 3. Data types for Algorithm 2.

```

nodetype * BumpUpOrInstall (ptrctr *fwd, ptrctr newifnull) {
    atomically {
        if (fwd->ptr == null)
            *fwd = newifnull; // value supplied for this case
        else
            *fwd = (fwd->ctr+1, fwd->ptr); // just bump up counter
        return fwd->ptr;
    }
}

nodetype * BumpDownOrRemove (ptrctr *fwd) {
    nodetype *removed;
    atomically {
        if (fwd->ptr == null) // no later nodes
            return null; // don't change anything, just return
        if (fwd->ctr == 1) { // I'm last one accessing from here
            removed = fwd->ptr; // remember node being removed
            *fwd = (0, null); // remove node
            return removed; // return removed node
        }
        *fwd = (fwd->ctr-1, fwd->ptr); // otherwise, decrement count
        return null; // no node removed
    }
}

```

Fig. 4. Atomic Specification of *BumpUpOrInstall* and *BumpDownOrRemove*.

erations that could be implemented as atomic instructions in hardware. We give the required semantics of these operations in Figure 4. We present our algorithms in terms of these atomic operations not because we expect or recommend that they will appear in real hardware, but to help guide any work on impossibility results that aim to address what wait-free mechanisms are possible for dynamic-sized data structures. In practice, these operations can be implemented in a lock-free manner using standard hardware synchronization support such as CAS; we present simple CAS-based lock-free implementations in Figure 5.

The code for Algorithm 2, presented in terms of *BumpUpOrInstall* and *BumpDownOrRemove* operations described above, appears in Figure 6. As in Algorithm 1, the *Store* operation is implemented as a single store instruction. In a system that provides only CAS to atomically update the forward pointers and their associated hold counts, we can guarantee only lock-freedom for the

```

nodetype * BumpUpOrInstall (ptrctr *fwd, ptrctr newifnull) {
    ptrctr of, nf;
    while (true) {
        of = *fwd;                                // Read current value
        if (of.ptr == null)                        // if it's null, prepare to install ...
            nf = newifnull;                        // ... value supplied for this case
        else
            nf = (of.ctr+1,of.ptr);                // else prepare to bump up counter
        if (CAS(fwd,of,nf))                        // try to install new value
            return newfwd.ptr;                    // return installed value if successful
    }
}

nodetype * BumpDownOrRemove (ptrctr *fwd) {
    ptrctr of;
    while (true) {
        of = *fwd;                                // read forward pointer and counter
        if (of.ptr == null)                        // at the end of list
            return null;                          // just return null (serialize at read)
        if (of.ctr == 1)                          // if I'm the last one accessing from here
            if (CAS(fwd,of,(0,null)))             // try to remove node
                return of.ptr;                   // return removed node
        else
            if (CAS(fwd,of,(of.ctr-1,of.ptr)));   // otherwise, try to decrement count
            return null;                          // no node removed
    }
}

```

Fig. 5. Code for lock-free implementations of *BumpUpOrInstall* and *BumpDownOrRemove*.

other operations: an operation may be prevented from making progress by a series of other operations continually modifying the pointer and hold count it is trying to update. However, if the *BumpUpOrInstall* and *BumpDownOrRemove* operations are atomic, then all the operations are wait-free.³ As before, the time complexity of the *Register* operation is bounded by a function of the maximum activity level during its execution, and the time complexity of the *Deregister* operation is bounded by a function of the maximum activity level during the execution of its corresponding *Register* operation. The *Collect* operation takes time proportional to the number of nodes in the list, which is bounded by the historical maximum activity level, but, as mentioned above, may be lower. In particular, when the last node in the list is deregistered, it will be removed from the list and freed by the *Cleanup* procedure invoked by *Deregister* unless some other thread is in the midst of a *Register* or *Collect* operation and has incremented the hold count of the next-to-last node. If this other thread is registering a node, then the activity level of the object was high at some time during the execution of its *Register* operation. However, a *Collect* operation may prevent the last node from being removed from the list and then before it invokes *Cleanup* and removes and frees the last node, another *Collect* operation may begin and read all the way down the list, again preventing the last node from being removed. This scenario can be repeated,

³ In the case of the *Register* and *Collect* operations, this assumes that there is a finite bound on the activity level. We emphasize that the algorithm does not need to know this bound.

```

nodetype * Register() {
  nodetype *newnode = new nodetype;    // Allocate new node just in case
  newnode->owned = true;                 // It's mine, all mine...
  newnode->val = (?,false);              // Not a stored value; ? = don't care
  newnode->forward = (0,null);           // Terminate node

  nodetype *nd = Head;                  // Start at head of list
  while (true) {
    if (CAS(&nd->owned,false,true)) {   // Try to claim this node
      delete newnode;                   // Didn't use it
      return nd;                         // Return it if successful
    }
    newnode->back = nd;                  // Set up back pointer
    next = BumpUpOrInstall(&nd->forward,(1,newnode));
                                          // Move to next and bump up counter, or
                                          // install new node if there's none.
    if (next == newnode)                // I installed my node;
      return newnode;                   // return it
    nd = next;                           // Move on to next node
  }
}

Store(nodetype *nd,valuetype v) {
  nd->val = (v,true);                   // just store value with valid indicator
}

Deregister(nodetype *nd) {
  nd->val = (?,false);                  // so value won't get returned later
  nd->owned = false;                    // give up ownership
  Cleanup(nd->back);                     // bump down counters and remove nodes if necessary
}

valueset Collect() {
  valueset S = emptyset;                // no values so far
  nd = Head;                             // start at first node
  while (nd) {                            // search through all nodes
    v = nd->val;                            // get value
    if (v.valid)                           // if it's a valid value,
      S = S union {(nd, v.value)};         // add it to the set
    prev = nd;                              // remember last node for cleanup
    nd = BumpUpOrInstall(&nd->forward,(0,null)); // get next (if any) and bump counter
  }
  Cleanup(prev->back);                     // follow back pointers and cleanup
  return S;                                // return set of values collected
}

void Cleanup(nodetype *nd) {
  nodetype *removed;
  while (nd) {
    removed = BumpDownOrRemove(&nd->forward); // bump down outgoing counter,
                                              // remove pointer if it becomes zero

    if (removed != null)                    // nobody is accessing the node; delete it
      delete removed;                       // go to previous node, if any
    nd = nd->back;
  }
}

```

Fig. 6. Algorithm 2.

so a series of overlapping *Collect* operations can indefinitely prevent any nodes from being freed. Thus, unless there is a point at which no *Collect* operation is executing, we cannot guarantee that space consumption of the collect object will decrease.

We can eliminate the problem of overlapping *Collect* operations preventing

unclaimed nodes at the end of the list from being cleaned up by separating out the hold count into the contribution by *Register* operations and the contribution by *Collect* operations. In the modified algorithm, a *Collect* operation detects the situation in which the contribution due to *Register* operations to the hold count it is accessing is 0. In this case, it does not go further down the list because there are no registered addresses further down the list. Therefore, this *Collect* operation does not increment subsequent hold counts in the list, so the scenario outlined earlier cannot occur. (Note that these changes would require modified versions of the *BumpDownOrRemove* and *BumpUpOrInstall* operations; this is straightforward given the code in Figures 4 through 6 and the description here.)

The adaptivity properties of Algorithm 2 (revised as described above, and assuming the *BumpDownOrRemove* and *BumpUpOrInstall* operations are atomic), are determined using the same proof technique used for Algorithm 1, shown in the appendix. Briefly, the time complexity of the *Register* operation is bounded by a function of the maximum activity level during its execution (as before), and the time complexity of the *Collect* operation is bounded by a function of the maximum activity level at any time during its execution or the execution of any *Register* operation that overlaps its execution or returns an address that is still registered during its execution. This also bounds the space consumption. Thus, we have removed any dependence on historical measures from both time complexity and space consumption.

The reason for the dependence of the time complexity of *Collect* on overlapping *Register* operations is discussed in Section 6, as are our efforts to further improve upon this property.

5 Other Applications

Solutions to the Collect problem are used as building blocks in many applications in concurrent computing, and thus solutions with better properties can directly improve properties of such applications [6]. We do not discuss such applications further. Instead, in this section, we discuss two applications of the *techniques* used in our algorithms; it is straightforward to see in each case how our algorithms can be adapted for these purposes.

5.1 Memory Management for Nonblocking Data Structures

Our first application is the one that originally motivated us to work on this problem, namely memory management for nonblocking dynamic-sized shared data structures. In our previous work in this area, we recently posed and solved the Repeat Offender Problem (ROP) [11]. Here we discuss only the details of this problem and our solution that are relevant to the work presented here; the reader is referred to previous paper for other details.

The Repeat Offender Problem requires threads to be able to dynamically

acquire and release locations, to be able to store values in these locations, and to be able to iterate over all values that have been stored in locations that have not subsequently been released. (In the parlance of [11], threads must be able to “hire” and “fire” “guards”, “post” guards on “values”, and determine which values are guarded in order to “liberate” those that are not.) In [11], we simplify the presentation of our solution by assuming that we know in advance an upper bound on the number of guards simultaneously employed, and allocating space for this number of guards as an array. This makes it straightforward to hire and fire guards: hiring is achieved by attempting to atomically claim each one in order until success (this is just the test-and-set-based renaming algorithm of Moir and Anderson [16]; it is proved there that we will not “fall off” the end of the array provided the assumption about the maximum number of names (guards) is not violated). While this is simple, it does cause some serious limitations. In particular, we must estimate the number of guards conservatively to ensure that we allocate enough. As a result, for example, our dynamic-sized lock-free FIFO queue implementation [10], which we achieved by applying our ROP solution to the population-oblivious algorithm of Michael and Scott [15], is population-aware. Similar recent results of Michael [14] have the same problem.

The algorithms presented in this paper can be used to overcome the above-described shortcoming of the simplified presentation of our ROP solution. In particular, *Register* can be used to implement the *HireGuard* operation, *Deregister* can be used to implement the *FireGuard* operation, *Store* can be used to implement the *PostGuard* operation, and *Collect* can be easily adapted to implement the functionality of the *Liberate* operation presented in our memory management paper [11].

The resulting ROP solutions will have properties corresponding to those for the particular Collect solution used. We believe that using the ROP solution achieved by applying Algorithm 2 as described above with the techniques described in [10] results in the first population-oblivious, space-adaptive implementation of a lock-free object that cannot be prevented from future memory reclamation by thread failures.

5.2 Long-Lived Renaming

As described by Attiya, et al. [6] and elsewhere, solutions to the Collect problem have been used in various solutions to the renaming problem [1,5,16]. However, all such renaming solutions are complicated, expensive, and population-aware, and are not space-adaptive. The techniques presented in this paper can be trivially adapted to solve the renaming problem much more efficiently, and in a space-adaptive, population-oblivious manner. The reason is that, in all of our algorithms, nodes are added only at the end of the list, and a node is not removed from the list while it still has successor nodes. Thus, we can solve the renaming problem simply by counting the number of iterations of the main

loop of *Register* in order to determine the number of nodes before the node eventually claimed, and taking this number as a name. With this approach, the renaming solution inherits all of the properties of the particular Collect solution that is adapted. (Note that no *Collect* operation is required for this application, so Algorithm 2 suffices without the modifications described at the end of Section 4.2.) This results in the first population-oblivious, space-adaptive, nonblocking renaming solutions.

6 Concluding Remarks

We have presented simple techniques based on widely available hardware synchronization primitives for designing nonblocking algorithms that do not require advance knowledge of the number of threads that participate, that have both time complexity and space consumption that adapt to various measures rather than being based on predefined worst-case scenarios, and that cannot be prevented from future memory reclamation by thread failures. We have presented these techniques in the context of various solutions to the Collect problem and have also described how these same techniques can be applied to achieve new algorithms with similar properties for solving the renaming problem, and for supporting memory management in dynamic-sized lock-free data structures. We have presented basic techniques; many variations, optimizations, and other applications are possible. It is particularly interesting to note that the “hold count” technique is not limited to lists; it can be applied to any structure in which each node has an in-degree of at most 1, and its predecessor (if any) can be found from the node.

To our knowledge, Algorithm 2 is the first truly dynamic-sized nonblocking algorithm that cannot be prevented from future memory reclamation by thread failures. However, a failed thread can prevent *some* memory from being reclaimed. In particular, if a thread fails after completing a *Register* operation and before its subsequent *Deregister*, then none of the nodes on the path from the head to the node registered by that thread will ever be reclaimed. This also explains why the time complexity of the *Collect* operation in Algorithm 2 depends on the activity level during *Register* operations that overlap with the *Collect* operation or whose corresponding *Deregister* operations have not completed before the *Collect* operation begins. (It is important to note that these “locked in” nodes can continue to be reused by subsequent *Register* operations, and that current and future nodes that are further from the head than this node can continue to be reclaimed.) While it is unavoidable (without sophisticated operating system support) that each thread failure can prevent the reclamation of a certain amount of memory, it is certainly desirable to minimize this amount, and also to prevent the wasted space from continuing to influence the time complexity of future operations. We are working on an algorithm that reduces to a constant the amount of memory that cannot be reclaimed due to each thread failure, and does not allow thread failures to

permanently affect *Collect* time complexity.

References

- [1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–104, 1999.
- [2] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 538–547, 1995.
- [3] O. Agesen, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS-based concurrent dequeues. *Theory of Computing Systems*, 2002. To appear. A preliminary version appeared in the Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures.
- [4] J. Anderson and M. Moir. Using local-spin k -exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:1–20, 1997. A preliminary version appeared in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141-150.
- [5] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 337–346, 1987.
- [6] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 2002. to appear.
- [7] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent dequeues. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.
- [8] D. Detlefs, P. Martin, M. Moir, and G. Steele. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.
- [9] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *ACM SIGPLAN international symposium on Memory management*. ACM Press, 2002.
- [10] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. Technical Report TR-2002-110, Sun Microsystems Laboratories, 2002.
- [11] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002. A improved version of this paper is in preparation for journal submission; please contact authors.

- [12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.
- [13] P. Martin, M. Moir, and G. Steele. Dcas-based concurrent dequeues supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.
- [14] M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Distributed Computing*, 2002.
- [15] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 267–276, 1996.
- [16] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995. A preliminary version appeared in *Proceedings of the 8th International Workshop on Distributed Algorithms*, 1994, pp. 141-155.
- [17] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus — making resilient algorithms fast in practice. In *Proceedings of the second annual ACM-SIAM Symposium on Discrete algorithms*, pages 351–362, 1991.
- [18] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

A Adaptivity Proof

The key adaptivity properties of our algorithms (in both time and space) are defined in terms of the activity level, as stated earlier.

Definition A.1 The *activity level* at some point in time is the sum of the number of addresses registered and the number of threads executing either *Register* or *Deregister* at that time.

Here we present a proof that *Register* operations in Algorithm 1 access the k th node in the list only if the activity level was at least k at some point during its execution. (We consider the first node to be node 1.) From this property, it is easy to see that the time complexity of the *Register* operation adapts to operation activity level, and that space consumption and the time complexity of the *Collect* operation adapt to the maximum historical activity level.

The main lemma of our proof is presented below. First, we need a couple of definitions.

Definition A.2 A thread p is *trying to claim the k th node* if it is executing a *Register* operation, has attempted and failed to CAS (i.e., modify using CAS)

the **owned** bit of the $(k - 1)$ -st node to *true* (if $k > 1$), and has not yet tried to CAS the **owned** bit of the k th node, or it is trying to add its (already claimed) new node as the next node of the $(k - 1)$ -st node.

Definition A.3 We use $(j, k]$ to denote the set of integers that are greater than j and at most k .

Lemma A.4 *For any thread p , if p tries to claim the k th node, then at some point during the execution of this Register operation, the number of claimed nodes, k th or earlier, plus the number of threads (including p) trying to claim the k th or an earlier node is at least k .*

Proof. The lemma follows directly from Lemma A.5, which is proved next. Lemma A.5 is strengthened slightly to allow a proof by induction on k . \square

Lemma A.5 *Consider any finite execution of Algorithm 1 such that p is trying to claim the k th node in the final state s^* of the execution. Let s_0 be the state immediately after p begins its current Register operation. For each state in the execution, consider the sum of the number of k th or earlier nodes that are claimed and the number of threads executing a Register operation that will claim a k th or earlier node before s^* . Then there is some state s between (in the execution) s_0 and s^* (inclusive) such that this sum is at least k .*

Proof. For $k = 1$, the claim holds because p is trying to claim the first node.

Suppose the claim holds for all j , $1 \leq j < k$. By the algorithm, p tried and failed to claim the j th node for each $j < k$ at some point after s_0 . It failed because the node was already claimed when p tried to claim it. We partition the $k - 1$ nodes before the k th into three groups:

- (i) those that were claimed continuously by some thread from s_0 until p attempts and fails to CAS their **owned** bits.
- (ii) those that were last claimed in the interval between s_0 and when p attempted to CAS their **owned** bit by some Register operation that started before s_0 .
- (iii) those that were last claimed in the interval between s_0 and when p attempted to CAS their **owned** bit by some Register operation that started after s_0 .

Note that, because each Register operation successfully claims at most one node, for every node in the second group, there is a distinct thread (not p) trying to claim the k th or earlier node at s_0 . Thus, if all the $k - 1$ nodes fall into the first two groups then at s_0 , the sum of the number of claimed nodes and the number of threads trying to claim the k th or earlier node is at least k (including p), so we are done.

Otherwise, the third group is nonempty. Choose j to be the maximum such that the j th node is in the third group, and let s' be the state immediately before the **owned** bit of the j th node is last set before being read by p . Because $j < k$, by the induction hypothesis, there is some state s (before s') such that

the number of nodes, j th or earlier, claimed in s plus the number of threads registering in s that will claim a j th or earlier node is at least j . Because the j th node is in the third group, s is between s_0 and s^* . We claim that s satisfies the condition for k ; that is, the sum of the number of claimed nodes between the first and k th inclusive in s and the threads executing *Register* operations in s that in some state between s and s^* successfully claim the k th or earlier node is at least k . To prove this, it suffices to show that the number of claimed nodes in $(j, k]$ in state s plus the number of threads registering in s that in some state between s and s^* successfully claim a node in $(j, k]$ is at least $k - j$.

If all nodes in $(j, k]$ are claimed in s , then we are done (as there are $k - j$ of them). Otherwise, for each such node that is not claimed, we know by the algorithm that it is claimed before p attempts to claim it. By our choice of j , these nodes cannot be in the third group, so they must be in the second group. Thus, each such node is claimed by a *Register* operation that started before s_0 , is still executing at s , and will claim that node before s^* . So for each unclaimed node in $(j, k]$, there is a corresponding thread registering in s that will claim that node between s and s^* , as required. \square