A Methodology for Implementing Highly Concurrent Data Structures

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Center
One Kendall Square
Cambridge, MA 02139

Abstract

A *concurrent object* is a data structure shared by concurrent processes. Conventional techniques for implementing concurrent objects typically rely on *critical sections*: ensuring that only one process at a time can operate on the object. Nevertheless, critical sections are poorly suited for asynchronous systems: if one process is halted or delayed in a critical section, other, non-faulty processes will be unable to progress. By contrast, a concurrent object implementation is *non-blocking* if it always guarantees that some process will complete an operation in a finite number of steps, and it is *wait-free* if it guarantees that *each* process will complete an operation in a finite number of steps. This paper proposes a new methodology for constructing non-blocking and wait-free implementations of concurrent objects. The object's representation and operations are written as stylized sequential programs, with no explicit synchronization. Each sequential operation is automatically transformed into a non-blocking or wait-free operation using novel synchronization and memory management algorithms. These algorithms are presented for a multiple instruction/multiple data (MIMD) architecture in which n processes communicate by applying *read*, *write*, and *compare&swap* operations to a shared memory.

1 Introduction

A *concurrent object* is a data structure shared by concurrent processes. Algorithms for implementing concurrent objects lie at the heart of many important problems in concurrent systems. Conventional techniques for implementing concurrent objects typically rely on *critical sections*: ensuring that only one process at a time can operate on the object. Nevertheless, critical sections are poorly suited for asynchronous systems: if one process is halted or delayed in a critical section, other, faster processes will be unable to progress. Possible sources of unexpected delay include page faults, exhausting one's scheduling quantum, preemption, and halting failures.

By contrast, a concurrent object implementation is *non-blocking* if it guarantees that some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. An implementation is *wait-free* if it guarantees that *each* process will complete an operation in a finite number of steps. The non-blocking condition permits individual processes to starve, but it guarantees that the system as a whole will make progress despite individual halting failures or delays. The wait-free condition does not permit starvation; it guarantees that all non-halted processes make progress. The non-blocking condition is appropriate for systems where starvation is unlikely, while the (stronger) wait-free condition is appropriate when some processes are inherently slower than others, as in some heterogeneous architectures. Either condition rules out the use of critical sections, since a process that halts in a critical section can force other processes trying to enter that critical section to run forever without making progress.

In this paper, we propose a new methodology for constructing non-blocking and wait-free implementations of concurrent objects. The object's representation and operations are written as stylized sequential programs, with no explicit synchronization. Each sequential operation is automatically transformed into a non-blocking or wait-free operation via a collection of novel synchronization and memory management algorithms introduced in this paper. We focus on a multiple instruction/multiple data (MIMD) architecture in which n processes communicate by applying *read*, *write*, and *compare&swap* operations to a shared memory. The

compare&swap operation is shown in Figure 1. We chose *compare&swap* for two reasons. First, it has been successfully implemented, having first appeared in the IBM System/370 architecture [20] [1]. Second, most other "classical" primitives are provably inadequate — we have shown elsewhere [18] that it is impossible [2] to construct non-blocking or wait-free implementations of many simple and useful data types using any combination of *read*, *write*, *test&set*, *fetch&add* [13], and memory-to-register *swap*. The *compare&swap* operation, however, is *universal* — at least in principle, it is powerful enough to transform any sequential object implementation into a non-blocking or wait-free implementation.

```
compare&swap(w: word, old, new: value)
    returns(boolean)
    if w = old
      then w := new
           return true
      else return false
      end if
    end compare&swap
```

Figure 1: The Compare&Swap Operation

Although we do not present specific language proposals in this paper, we believe the methodology introduced here lays the foundation for a new approach to programming languages for shared-memory multiprocessors. As illustrated by Andrews and Schneider's comprehensive survey [1], most language constructs for shared memory architectures focus on techniques for mutual exclusion and scheduling. Only recently has attention started to shift to models that permit a higher degree of concurrency [18, 19, 32]. Because our methodology is based on automatic transformation of sequential programs, the formidable problem of reasoning about concurrent data structures is reduced to the more familiar domain of sequential reasoning. As discussed below in the section on related work, the transformed implementations are simpler and more efficient, both in time and space, than earlier constructions of this kind. For example, the concurrent priority queue example in Section 4.3 is an interesting algorithm in its own right.

In Section 2, we give a brief survey of related work. Section 3 describes our model. In Section 4, we present protocols for transforming sequential implementations into non-blocking implementations. To illustrate our methodology, we present a novel non-blocking implementation of a *skew heap* [35], an approximately-balanced binary tree used to implement a priority queue. In Section 5, we show how to transform our non-blocking protocols into wait-free protocols. Section 6 summarizes our results, and concludes with a discussion. Since rigorous models and proofs are beyond the scope of a paper of this length, our presentation here is deliberately informal, emphasizing the intuition and motivation underlying our algorithms.

[1] The System/370's *compare&swap* returns the register's previous value in addition to the boolean condition.

[2] Although our impossibility results were presented in terms of wait-free implementations, they hold for non-blocking implementations as well.

2 Related Work

Early work on non-blocking protocols focused on impossibility results [6, 7, 8, 9, 11, 18], showing that certain problems cannot be solved in asynchronous systems using certain primitives. By contrast, a synchronization primitive is *universal* if it can be used to transform any sequential object implementation into a wait-free concurrent implementation. The author [18] gave a necessary and sufficient condition for universality: a synchronization primitive is universal in an n-process system if and only if it solves asynchronous consensus [11] for n processes. Although this result showed that wait-free (and non-blocking) implementations are possible in principle, the universal construction was too inefficient to be practical. Plotkin [32] gives a detailed universal construction for a *sticky-bit* primitive. This construction, while more efficient than the consensus-based construction, is still not entirely practical, as each operation may require multiple scans of all of memory. The universal constructions presented here are simpler and more efficient than earlier constructions, primarily because *compare&swap* seems to be a "higher-level" primitive than sticky-bits.

Many researchers have studied the problem of constructing wait-free *atomic read/write registers* from simpler primitives [4, 5, 22, 25, 28, 30, 31, 34]. Atomic Registers, however, have few if any interesting applications for concurrent data structures, since they cannot be combined to construct non-blocking or wait-free implementations of elementary data types such as queues, directories, or sets [18]. There exists an extensive literature on concurrent data structures constructed from more powerful primitives. Gottlieb et al. [14] give a highly concurrent queue implementation based on the *replace-add* operation, a variant of *fetch&add*. This implementation permits concurrent enqueuing and dequeuing processes, but it is blocking, since it uses critical sections to synchronize access to individual queue elements. Lamport [24] gives a wait-free queue implementation that permits one enqueuing process to execute concurrently with one dequeuing process. Herlihy and Wing [17] give a non-blocking queue implementation, employing *fetch&add* and *swap*, that permits an arbitrary number of enqueuing and dequeuing

processes. Lanin and Shasha [26] give a non-blocking set implementation that uses operations similar to *compare&swap*. There exists an extensive literature on locking algorithms for concurrent B-trees [2, 27, 33] and for related search structures [3, 10, 12, 15, 21]. Our concurrent skew heap implementation in Section 4.3 uses *futures*, a form of lazy evaluation used in MultiLisp [16].

3 Model

In this section we give an informal presentation of our model, focusing on the intuition behind our definitions. A more formal treatment appears elsewhere [17].

A *concurrent system* consists of a collection of n sequential *processes* that communicate through shared typed *objects*. Processes are sequential — each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. We make *no* fairness assumptions about processes. A process can halt, or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

Objects are data structures in memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to manipulate that object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time by a single process. For example, the behavior of a queue object can be specified by requiring that *enq* insert an item in the queue, and that *deq* remove the oldest item present in the queue. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions.

An object is *linearizable* [17] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of non-overlapping operations is preserved. As discussed in more detail elsewhere [17], the notion of linearizability generalizes and unifies a number of *ad-hoc* correctness conditions in the literature, and it is related to (but not identical with) correctness criteria such as sequential consistency [23] and strict serializability [29]. We use linearizability as the basic correctness condition for all concurrent objects constructed in this paper.

A natural way to measure the time complexity of a non-blocking implementation is the *system latency*, defined to be the largest number of steps the system can take without completing an operation. For a wait-free implementation, the *process latency* is the largest number of steps a process can take without completing an operation. Both kinds of latency measure worst-case performance. We are usually interested in latency as a function of the system size. For brevity, we say that an implementation is $O(n)$ *non-blocking* (*wait-free*) if it has $O(n)$ system (process) latency. Note that an implementation using critical sections has infinite system and process latencies, since the system can take an arbitrary number of steps without completing an operation if some process is delayed in a critical section.

Our methodology is based on the following steps.

1. The *programmer* chooses a representation for the object, and implements a set of *sequential operations*. The sequential operations are written in a conventional sequential language, with no explicit synchronization. They are subject to the following important restriction: they must written in a *functional* style — an operation that changes the object state is not allowed to modify the object in place, instead it must compute and return a (logically) distinct version of the object.

2. Using the synchronization and memory management algorithms described below, each sequential operation is transformed into a *non-blocking* (or *wait-free*) *operation*. (The transformed operations typically appear to update the object in place.)

This transformation could be done by a compiler.

For example, to implement an operation with the following signature:

```
operation(x: object, a: value)
        returns(value)
```

the programmer would implement a sequential operation with the following signature:

```
OPERATION(x: object, a: value)
        returns(object, value)
```

By convention, names of sequential operations appear in small capitals, and names of non-blocking operations in lower-case.

4 Non-Blocking Protocols

4.1 Single Word Objects

We first consider objects whose values fit in a single word of memory. The sequential operation is transformed into a non-blocking operation by the *Single Word Protocol*, shown in Figure 4.1. Here, we show a sequential *fetch&add* operation, together with its non-blocking transformation. Each process reads the object's current value into a local variable, calls the sequential operation to compute a new value, and then attempts to reset the object to that new value using

compare&swap. If the *compare&swap* succeeds, the operation returns; otherwise the loop is resumed. We use the following terminology. Each execution of the loop body is an *attempt*. An attempt *succeeds* when the *compare&swap* returns *true*, otherwise it *fails*. An interval between reading the object value and the next *compare&swap* is called a *window*.

```
FETCH&ADD(x: integer, v: integer)
   returns(integer, integer)
   return (x+v, x)
   end FETCH&ADD

fetch&add(obj: object, v: integer) returns(integer)
   success: boolean := false
   loop exit when success
        old: integer := obj
        new: integer, r: value := FETCH&ADD(old, v)
        success := compare&swap(x, old, new);
        end loop
   return r
   end fetch&add
```

Figure 2: The Single Word Protocol

The Single Word Protocol is $O(n)$ non-blocking. Process P's *compare&swap* will fail only if another process succeeds during P's window. At worst, a successful *compare&swap* can force the next $n-1$ *compare&swap* operations to fail, but n-th *compare&swap* will then succeed.

4.2 Small Objects

We now consider objects that span multiple words. A *block* is a fixed-length contiguous sequence of words small enough to be copied efficiently. A *small object* is one that occupies a single block. As before, the programmer writes sequential operations in a functional style. Instead of returning a new object value, however, a sequential operation returns *a pointer to* a newly-allocated block holding the object's new value.

The programmer is given the following primitives for memory management: The *alloc* procedure returns a pointer to a newly-allocated block, and the *free* procedure indicates that a block is no longer in use. (Here we use "T*" as shorthand for "pointer to T".) A sequential operation modifies a small object by allocating a new block, initializing it, freeing the old block, and returning the new block. For example, Figure 3 shows a sequential POP operation for a fixed-size stack. If the stack is empty, the operation leaves the stack unchanged and returns an exception value; otherwise, it allocates a new block, copies the remaining items, frees the old block, and returns the new block and popped item.

```
% The stack representation:
stack = record[size: integer, data: array[item]]

POP(s: block*) returns(block*, item)
   if s.size = 0
     then return (s, "empty")
     end if
   r: item := s.data[s.size]
   s' := alloc()
   s'.size := s.size - 1
   for i in 1..s'.size do
       s'.data[i] := s.data[i]
       end for
   free(s)
   return (s', r)
   end POP
```

Figure 3: A Sequential POP Operation

At run time, each process keeps the following information in global variables: *root* is a pointer to the word holding the object's root pointer, *old* is the root pointer's value when the current attempt started, *frozen_list* is a list of the blocks read during the current attempt, *commit_set* and *abort_set* are respectively the sets of blocks freed and allocated during the current attempt, and *pool* is a set of n blocks from which new blocks are allocated. The *frozen_list*, *commit_set*, *abort_set*, and *pool* data structures are private to each process and require no synchronization.

```
freeze(b: block*) returns(block*) signals(abort)
   append(frozen_list, b)
   inc(b.readers)
   if root ≠ old
     then signal abort
     else return b
     end if
   end freeze

unfreeze(b: block*)
   dec(b.readers)
   end unfreeze
```

Figure 4: The Freeze and Unfreeze Procedures

Each block has a *readers* field, which is an integer that counts the number of processes currently reading that block. A block will not be reallocated and reused as long as its *readers* field is non-zero. The *readers* field is manipulated by non-blocking *inc* and *dec* operations implemented by the Single Word Protocol. Before a process P can read a block, it must apply the *freeze* procedure in Figure 4. This procedure appends the block to P's *frozen_list* and increments the *readers* count. If

the object's root pointer has changed since the start of the current window, the procedure raises an **abort** exception, described in more detail below, indicating that the current attempt has failed. This check is necessary to avoid the race condition in which another process reallocates and reinitializes the block after P reads the pointer but before it increments the *readers* field. (This check also gives an "early warning" that the current attempt is doomed to failure.) The *unfreeze* procedure simply decrements the block's *readers* field.

The *alloc* and *free* procedures are shown in Figure 5. To allocate a block, P simply scans its pool until it finds a block whose *readers* field is zero. (Because each process has at most one block frozen, one can show that P's pool of n blocks must contain at least one block with a zero *readers* count.) P increments the *readers* count to inhibit reallocation, and then inserts the block in its *abort_set*. To free a block, a process simply inserts it in its *commit_set*.

```
alloc() returns(block*)
   for b: block* in elements(pool) do
      if b.readers = 0
        then inc(b.readers)
             append(abort_set, b)
             return b;
        end if
      end for
   end allocate


free(b: block*)
   append(commit_set, b)
   end free
```

Figure 5: The Pool Management Protocols

We are now ready to examine the *Small Object Protocol* in Figure 6. The object is represented by a *root* pointer that points to the block holding the object's current *version*. Each time through the loop, the process freezes the object's current version (Statement #1), calls the sequential operation to create a new version (#2). If a pointer comparison indicates that the operation changed the object's state (#3), the process calls *compare&swap* to attempt to "swing" the pointer from the old version to the new. P unfreezes the blocks it has frozen [3]. (The **abort** exception in the *freeze* procedure transfers control here.) If the *compare&swap* succeeded (#4), P unfreezes the blocks in its *commit_set*, otherwise it unfreezes the blocks in its *abort_set*. Finally, P resets its private data structures (#5).

[3] In the Small Object Protocol, the *frozen_list*, *commit_set*, and *abort_set* contain at most one block, but they will contain more in subsequent versions of the protocol.

```
pop(root: block*) returns(value)
   success: boolean := false
   loop exit when success
1:      old: block* := freeze(root)
2:      new: block*, res: value := POP(old)
        success := true
3:      if old ≠ new
          then success :=
                 compare&swap(obj, old, new)
        end if
abort:  unfreeze blocks in frozen_list
4:      if success
          then unfreeze blocks in commit_set
          else unfreeze blocks in abort_set
        end if
5:      reset frozen_list, commit_set and abort_set
        end loop
   return result
   end pop
```

Figure 6: The Small Object Protocol

The Small Object Protocol is $O(n^2)$ non-blocking. In the worst-case scenario, an adversary scheduler can force the processes to execute $O(n^2)$ *compare&swap* operations to complete n increments before it allows the first *compare&swap* to be applied to the root pointer. If the architecture provides a unit-time *fetch&add*, then the Small Object Protocol is $O(n)$ non-blocking, just like the Single-Word Protocol. Since each process keeps a pool of n blocks, a population of K small objects requires a total of $K + n^2$ blocks.

4.3 Large Objects

In this section, we show how to extend the previous section's protocols to objects that are too large to be copied all at once.

A *large object* is represented by a set of blocks linked by pointers. As before, the object's operations are implemented by sequential programs written in a functional style. The advantage of splitting the object into multiple blocks is that an operation that modifies the object need copy only those blocks that have changed. In the example below, inserting or deleting an element from a tree typically leaves most of the tree changed.

As before, a process must freeze each block before reading it. If the freeze fails, the **abort** exception causes a non-local transfer of control to the statement immediately after the *compare&swap* in the main loop. Freezing the same block more than once does not jeopardize correctness, since the *frozen_list* can contain duplicates, and each *freeze* will have a matching *unfreeze*. Calls to *freeze* can be inserted automatically by a compiler whenever a sequential operation copies a block pointer from a block to a local variable.

Let m be the maximum number of blocks a process can freeze in the course of an operation. If each process keeps $(n-1)m+1$ blocks in its pool, then *alloc* will always find a block with a zero *readers* count. Since large objects encompass a variable number of blocks, however, a process may find it has fewer or more blocks in its pool after an operation. Once a block's *readers* count reaches zero, it can be made accessible to other processes by placing it in a *shared pool*. The shared pool consists of n free lists, one for each process. Each list always holds at least one block. Only process P can place blocks on list P, but any process can remove a block from any list. To avoid "hot-spot" contention, each process attempts to allocate blocks from its own free list; only when that list is exhausted does it try to "steal" blocks from other lists.

The shared pool is represented by two arrays of pointers: *first* points to the first block in each list, and *last* points to the last block. These pointers are always non-nil. Each block has a *next* field that points to the next block in the list. (The *next* field can be overwritten once the block is allocated.) A process removes a block from the shared pool using *shared_alloc* (Figure 7). It scans the *first* array, starting with its own list. For each list, the process "freezes" the first block in the usual way: it reads the *first* pointer, increments the block's *readers* count, and then rereads the *first* pointer. If the *first* pointer has changed, then the block has already been allocated, so the process decrements the block's *readers* count and restarts. While the block is frozen, it will not be returned to the shared free pool, so it is safe to read the block's next pointer. If this pointer is nil, then the list contains only one block (which cannot be removed) so the process decrements the *readers* count and moves on to the next list. If the next pointer is non-nil, the process tries to "swing" the *first* pointer using a *compare&swap*. Since an allocation can fail only if a concurrent allocation succeeds, this protocol is $O(n)$ non-blocking.

When a block's *readers* count reaches zero, a process may transfer it from its private pool to the shared pool. The process calls the *shared_free* procedure (Figure 7), which simply sets the next field of the last block to the new block, and then updates the *last* pointer. Since no other process manipulates the *last* pointer or allocates the last block, this procedure is $O(1)$ wait-free.

```
% Code for Process P
shared_alloc() returns(block*)
    who: integer := P
    loop
        ok: boolean := false
        loop
            old: block* := first[who]
            inc(old.readers)
            if first[who] ≠ old
              then goto abort
              end if
            new: block* := old.next
            if new = nil
              then dec(old.readers)
                   break
              end if
            ok := compare&swap(first[who], old, new)
abort:      dec(old.readers)
            if ok
              then return old
              end if
            end loop
        who := (who + 1) mod n
        end loop
    end shared_alloc

shared_free(b: block*)
    last[P].next := b
    last[P] := b
    end shared_free
```

Figure 7: Manipulating the Shared Free List

4.4 A Non-Blocking Skew Heap

A *priority queue* is a set of items taken from a totally-ordered domain, providing two operations: *enq* inserts an item into the queue, and *deq* removes and returns the least item in the queue. One way to implement a priority queue is with a *skew heap* [35], an approximately-balanced binary tree in which each node stores an item, and each node's item is less than or equal to any item in the subtree rooted at that node. The amortized cost of enqueuing or dequeuing an item in a skew heap is logarithmic in the size of the tree. For our purposes, the advantage of a skew heap is that update operations leave most of the tree nodes untouched.

A sequential skew heap implementation appears in Figure 8. (For brevity, we assume calls to *freeze* are inserted by the compiler.) The *meld* operation merges two heaps. It chooses the heap with the lesser root, swaps its right and left children (for balance), and then melds the right child with the other heap. To insert item x in h, *enq* melds h with the heap containing x alone. To remove an item from h, *deq* removes the item at the root and melds the root's left and right subtrees. If the queue is empty, *deq* returns an exception value.

An advantage of the Large Object Protocol is that the programmer who implements the sequential operations can exploit type-specific properties to do as little copying as possible. The skew heap implementation does logarithmic amortized copying, while the Small Object Protocol would do linear copying, since it would copy

```
heap = record[value: item, left: heap*, right: heap*]

ENQ(h: heap*, x: item) returns(heap*)
   h': heap* := alloc()
   h'.value, h'.left, h'.right := x, nil, nil
   return MELD(h, h')
   end ENQ

DEQ(h: heap*) returns(heap*, item)
   if h = nil
     then return (h, "empty")
     end if
   r: value := h.value
   h': heap* := MELD(h.left, h.right)
   free(h)
   return (h', r)
   end DEQ

MELD(h1, h2: heap*) returns(heap*)
   select
     case h1 = nil do return h2
     case h2 = nil do return h1
     end select
   if h2.value < h1.value
     then h1, h2 := h2, h1
     end if
   h := alloc()
   h.value := h1.value
   h.right := h1.left
   h.left := MELD(h1.right, h2)
   free(h1)
   return h
   end MELD
```

Figure 8: A Skew Heap

the entire heap. The Large Object Protocol is thus more efficient, not only because each operation does less work, but because each operation's window is shorter, implying that each attempt is more likely to succeed.

4.5 Benevolent Side-Effects

A *benevolent side-effect* is a change to the object's representation that has no effect on its abstract value. Examples of benevolent side-effects include rebalancing a tree, caching the most recently accessed key in a database, etc. An operation that carries out a benevolent side-effect may *checkpoint* the object at that point. A checkpoint will attempt to swing the root pointer to the new version. If the checkpoint succeeds, then the operation resumes, and the effects of the checkpoint persist even if the next *compare&swap* fails. If the checkpoint fails, then the operation must be restarted.

In the skew heap example, benevolent side-effects can be used to narrow the *enq* operation's window even further. A *future* [16] is a data structure representing an unevaluated expression. Here we use futures to postpone executing *meld* operations. A node in the heap is either a regular node (as before) or a *future* node. A *future* node has a special tag indicating that the node's value is absent, and must be reconstructed by melding its right and left subheaps. The FUTURE operation creates a future, and the TOUCH operation evaluates it (Figure 9). The modified skew heap implementation appears in Figure 10. The *enq* operation creates a future and returns immediately, *deq* touches its argument and checkpoints the result before removing the item, and *meld* creates a future instead of calling itself recursively. This use of futures makes *enq* windows shorter and *deq* windows longer, and the checkpoint splits the *deq* operation into smaller "chunks", reducing the work lost when a checkpoint or operation aborts.

```
FUTURE(right: heap*, left: heap*) returns(heap*)
   f: heap* := alloc()
   f.value, f.left, f.right := "future", left, right
   return f
   end FUTURE

TOUCH(h: heap*) returns(heap*, value)
   if h.value = "future"
     then h': heap* := MELD(h.right, h.left)
          free(h)
          return (h', h'.value)
     else return (h, h.value)
     end if
   end TOUCH
```

Figure 9: Procedures for Manipulating Futures

5 Wait-Free Protocols

There are three places in the Large Object Protocol where a process can starve: the *inc* and *dec* procedures in *freeze* and *unfreeze*, the main loop in the Large Object Protocol itself, and the *shared_alloc* procedure.

5.1 Freeze and Unfreeze

The *Inc* procedure in *Freeze* can be made wait-free simply by aborting the operation if the increment fails after $2(n-1)$ attempts. While the root pointer remains unchanged, an attempt to increment the *readers* field can be interrupted by $n-1$ "late" decrements and $n-1$ increments. If an increment fails after that, then the block must have been reallocated, the root pointer must have changed, and the operation should be restarted. The decrement in *Unfreeze* is already wait-free. A process will unfreeze a block only if the block's *readers* field is non-zero. While that field remains non-zero, the block will not be reallocated, and eventually no other

```
ENQ(h: heap*, x: item) returns(heap*)
  h': heap* := alloc()
  h'.value, h'.left, h'.right := x, nil, nil
  return FUTURE(h, h')
  end ENQ

DEQ(h: heap*) returns(heap*, item)
  if h = nil
    then return (h, "empty")
    end if
  h, r := TOUCH(h)
  checkpoint h
  h': heap* := MELD(h.left, h.right)
  free(h)
  return (h', r)
  end DEQ

MELD(h1, h2: heap*) returns(heap*)
  select
    case h1 = nil do return h2
    case h2 = nil do return h1
    end select
  h1, v1 := TOUCH(h1)
  h2, v2 := TOUCH(h2)
  if v2 < v1
    then h1, v1, h2, v2 := h2, v2, h1, v1
    end if
  h := alloc()
  h.value := h1.value
  h.right := h1.left
  h.left := FUTURE(h1.right, h2)
  free(h1)
  return h
  end MELD
```

Figure 10: A Skew Heap with Futures

processes will be trying to increment or decrement the count.

5.2 The Main Loop

The main loop can be made wait-free by having processes "help" one another, (c.f. [18, 19, 32]). When process P begins an operation, it allocates an *invocation block* with the following fields: *Op* is the operation being invoked, and *Args* is the invocation's arguments. P then "announces" it has a pending invocation by storing a pointer to its invocation block in an array *intent* shared by all processes. Even if P does not succeed in completing its invocation, some other process eventually will.

Instead of pointing directly to the object, the root pointer points to a *header block* with the following fields: *last* is a pointer to the invocation block for the last operation, *result* is the last invocation's result, *who* is the process that requested the invocation, *turn* is a mod n counter used to prevent starvation, and *version* is a pointer to the object itself.

The Wait-Free Protocol (Figure 11) works as follows. Process P creates and initializes an invocation block, stores it in *intent*[P], and enters the main loop (Statement #1), which it executes until its invocation block in *intent*[P] is replaced by a result. P freezes the object's header block and its last invocation block (#2 and #3). If the last invocation was carried out on behalf of process Q, then P "notifies" Q by trying to replace Q's invocation block in *intent* with the invocation's result (#4). P then looks for a process to help, checking the process named by header block's *turn* field (#5), and then itself (#6). If neither process has a pending invocation, P is finished (#7). Otherwise, P executes the operation (#8), and checks whether the object's state has changed (#9). If not, then P simply notifies Q that the invocation is complete. Otherwise, P allocates and initializes a new header block, and then calls *compare&swap* to swing the pointer to the new version (#10). As before, P then unfreezes the blocks it has frozen, and returns unused blocks to its pool.

The Wait-Free Protocol is $O(n^2)$ wait-free. The *turn* variable ensures that a process can execute the main loop at most $n+1$ times, and each time through the loop it can execute at most $O(n)$ *compare&swap* operations freezing blocks. If the architecture provides a unit-time *fetch&add* operation, then this protocol is $O(n)$ wait-free.

5.3 The Free List

The *shared_alloc* procedure can be made wait-free in a similar way. Process P "announces" its intention to allocate a block by storing a *nil* value in position P of a shared intentions vector. P loops until some process replaces that *nil* value with a pointer to a newly-allocated block. Each time through the loop, P checks its entry in the intentions vector. If that entry is still *nil*, P attempts to allocate a block. If the allocation succeeds, P scans the intentions array, starting after the last position it scanned in the previous iteration, and attempts to replace the next *nil* value with the block pointer. If P reaches the end of the array without disposing of the block, then it returns the extra block to the pool (*shared_free* is already wait-free) and returns to block allocated for it in the intentions array. This protocol is $O(n^2)$ wait-free.

6 Conclusions

We have introduced a new methodology for constructing highly-concurrent data structures from sequential implementations. This methodology makes the following contributions:

```
% Code for process P
op(root: word*, arg: value) returns(value)
   intent[P] := allocate and initialize invocation block
1: loop exit when typeof(intent[P]) = result
        success: boolean := true
2:      old: *block := freeze(root)
3:      last: *block := freeze(old.last)
4:      compare&swap(intent[old.who], last, old.result)
        help: invocation* := freeze(intent[old.turn])
        select
5:        case typeof(help) = invocation do
              Q := old.turn
6:        case typeof(intent[P]) = invocation do
              help := intent[P]
              Q := P
7:        otherwise do goto abort
          end select
8:      v: block*, r: value :=
          help.op(old.version, help.arg)
9:      if old.version = v
          then compare&swap(intent[Q], help, r)
          else new: block* := alloc()
               new.version := v
               new.result := r
               new.turn := (old.turn + 1) mod n
               new.last := help
               new.who := Q
10:            success := compare&swap(root, old, new)
          end if
abort:  unfreeze blocks in frozen_list
        if success
          then unfreeze blocks in commit_set
          else unfreeze blocks in abort_set
          end if
        reset frozen_list, commit_set and abort_set
        end loop
   return intent[P]
   end op
```

Figure 11: The Wait-Free Protocol

- Non-blocking and wait-free implementations are better suited to asynchronous shared-memory multiprocessors than conventional techniques that rely on critical sections.
- Because programmers write stylized sequential programs, formal and informal reasoning about correctness is greatly simplified.
- As illustrated by the concurrent skew heap example, our methodology permits programmers to exploit type-specific properties to make data structures more efficient.
- Our algorithms are based on a conservative architecture, using the well-known *compare&swap* operation as the only "powerful" primitive.
- The transformed implementations are efficient in time and space.

We are currently experimenting with a multiprocessor implementation of these algorithms.

Acknowledgments

I am grateful to Bill Weihl for pointing out a bug in an earlier version of this paper.

References

[1] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, 1983.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 1(1):1–21, 1977.

[3] J. Biswas and J.C. Browne. Simultaneous update of priority structures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 124–131, 1987.

[4] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.

[5] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[6] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[7] D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[8] C. Dwork, N. Lynch, and L Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, April 1988.

[9] C. Dwork, D. Shmoys, and L. Stockmeyer. Flipping persuasively in constant expected time. In *Twenty-Seventh Annual Symposium on Foundations of Computer Science*, pages 222–232, October 1986.

[10] C.S. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[11] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[12] R. Ford and J. Calhoun. Concurrency control mechanisms and the serializability of concurrent tree algorithms. In *3rd ACM Symposium on Princples of Database Systems*, pages 51–60, 1984.

[13] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer – designing an mimd parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[14] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[15] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th ACM Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[16] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Prog. Languages and Systems*, 7(4):501–538, 1985.

[17] M. Herlihy and J. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[18] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[19] M.P. Herlihy. Wait-free synchronization. Submitted for publication., July 1989.

[20] IBM. System/370 principles of operation. Order Number GA22-7000.

[21] D.W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, January 1989.

[22] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[24] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[25] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[26] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[27] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[28] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[29] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[30] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[31] G.L. Peterson and J.E. Burns. Concurrent reading while writing ii: the multi-writer case. Technical Report GIT-ICS-86/26, Georgia Institute of Technology, December 1986.

[32] S.A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–176, 1989.

[33] Y. Sagiv. Concurrent operations on b-trees with overtaking. In *ACM Princples of Database Systems*, pages 28–37, January 1985.

[34] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, 1987.

[35] D.D. Sleator and R.E. Tarjan. Self adjusting binary trees. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, pages 52–59, 1983.