Universal Constructions for Large Objects *

James H. Anderson and Mark Moir

Dept. of Computer Science, University of North Carolina at Chapel Hill

Abstract

We present lock-free and wait-free universal constructions for implementing large shared objects. Most previous universal constructions require processes to copy the entire object state, which is impractical for large objects. Previous attempts to address this problem require programmers to explicitly fragment large objects into smaller, more manageable pieces, paying particular attention to how such pieces are copied. In contrast, our constructions are designed to largely shield programmers from this fragmentation. Furthermore, for many objects, our constructions result in lower copying overhead than previous ones.

Fragmentation is achieved in our constructions through the use of *load-linked*, *store-conditional*, and *validate* operations on a "large" multi-word shared variable. Before presenting our constructions, we show that these operations can be efficiently implemented from similar one-word primitives.

1 Introduction

This paper extends recent research on *universal* lock-free and wait-free constructions of shared objects [3, 4]. Such constructions can be used to implement any object in a lock-free or a wait-free manner, and thus can be used as the basis for a general methodology for constructing highly-concurrent objects. Unfortunately, this generality often comes at a price, specifically space and time overhead that is excessive for many objects. A particular source of inefficiency in previous universal constructions is that they require processes to copy the entire object state, which is impractical for large objects. In this paper, we address this shortcoming by presenting universal constructions that can be used to implement large objects with low space overhead.

We take as our starting point the lock-free and wait-free universal constructions presented by Herlihy in [4]. In these constructions, operations are implemented using "retry loops". In Herlihy's lock-free universal construction, each process's retry loop consists of the following steps: first, a shared object pointer is read using a *load-linked* (LL) operation, and a private copy of the object is made; then, the desired operation is performed on the private copy; finally, a *store-conditional* (SC) operation is executed to attempt to "swing" the shared object pointer to point to the private copy. The SC operation may fail, in which case these steps are repeated. This algorithm is not wait-free because the SC of each loop iteration may fail. To ensure termination, Herlihy's wait-free construction employs a "helping" mechanism, whereby each process attempts to help other processes by performing their pending operations together with its own. This mechanism ensures

*Work supported, in part, by NSF contract CCR 9216421, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAHO4-95-1-0323.

that if a process is repeatedly unsuccessful in swinging the shared object pointer, then it is eventually helped by another process (in fact, after at most two loop iterations).

As Herlihy points out, these constructions perform poorly if used to implement large objects. To overcome this problem, he presents a lock-free construction in which a large object is fragmented into blocks linked by pointers. In this construction, operations are implemented so that only those blocks that must be accessed or modified are copied.

Herlihy's lock-free approach for implementing large objects suffers from three shortcomings. First, the required fragmentation is left to the programmer to determine, based on the semantics of the implemented object. The programmer must also explicitly determine how copying is done. Second, Herlihy's approach is difficult to apply in wait-free implementations. In particular, directly combining it with the helping mechanism of his wait-free construction for small objects results in excessive space overhead. Third, Herlihy's large-object techniques reduce copying overhead only if long "chains" of linked blocks are avoided. Consider, for example, a large shared queue that is fragmented as a linear sequence of blocks (i.e., in a linked list). Replacing the last block actually requires the replacement of every block in the sequence. In particular, linking in a new last block requires that the pointer in the previous block be changed. Thus, the next-to-last block must be replaced. Repeating this argument, it follows that every block must be replaced.

Our approach for implementing large objects is also based upon the idea of fragmenting an object into blocks. However, it differs from Herlihy's in that it is array-based rather than pointer-based, i.e., we view a large object as a long array that is fragmented into blocks. Unlike Herlihy's approach, the fragmentation in our approach is not visible to the user. Also, copying overhead in our approach is often much lower than in Herlihy's approach. For example, we can implement shared queues with constant copying overhead.

Our constructions are similar to Herlihy's in that operations are performed using retry loops. However, while Herlihy's constructions employ only a single shared object pointer, we need to manage a collection of such pointers, one for each block of the array. We deal with this problem by employing LL, SC, and *validate* (VL) operations that access a "large" shared variable that contains all block pointers. This large variable is stored across several memory words.[1] In the first part of the paper, we show how to efficiently implement them using the usual single-word LL, SC, and VL primitives. We present two such implementations, one in which LL may return a special value that indicates that a subsequent SC will fail — we call this a *weak*-LL — and another in which LL has the usual semantics. In both implementations, LL and SC on a W-word variable take $O(W)$ time and VL takes constant time. The first of these implementations is simpler than the second because weak-LL does not have to return a consistent multi-word value in the case of interference by a concurrent SC. Also, weak-LL can be used to avoid unnecessary work in universal algorithms (there is no point performing private updates when a subsequent SC is certain to fail). For these reasons, we use weak-LL in our universal constructions.

Our wait-free universal construction is the first such construction to incorporate techniques for implementing large objects. In this construction, we impose an upper bound on the number of private blocks each process may have. This bound is assumed to be large enough to accommodate any single operation. The bound affects the manner in which processes may help one another. Specifically, if a process attempts to help too many other processes simultaneously, then it runs the risk of using more private space than is available. We solve this problem by having each process help as many processes as possible with each operation, and by choosing processes to help in such a way that all processes

[1] The multi-word operations considered here access a *single* variable that spans multiple words. Thus, they are not the same as the multi-word operations considered in [1, 2, 5, 6], which access *multiple* variables, each stored in a separate word. The multi-word operations we consider admit simpler and more efficient implementations than those considered in [1, 2, 5, 6].

```
shared var X: record pid: 0..N − 1; tag: 0..1 end;
            BUF: array[0..N − 1, 0..1] of array[0..W − 1] of wordtype
initially X = (0, 0) ∧ BUF[0, 0] = initial value of the implemented variable V

private var curr: record pid: 0..N − 1; tag: 0..1 end; i: 0..W − 1; j: 0..1
initially j = 0

proc Long_Weak_LL(var r: array[0..W − 1]
                      of wordtype) returns 0..N
1:  curr := LL(X);
    for i := 0 to W − 1 do
2:      r[i] := BUF[curr.pid, curr.tag][i]
    od;
3:  if VL(X) then return N
4:  else return X.pid fi

proc Long_SC(val: array[0..W − 1] of wordtype)
                                    returns boolean
4:  j := 1 − j;
    for i := 0 to W − 1 do
5:      BUF[p, j][i] := val[i]
    od;
6:  return SC(X, (p, j))
```

Figure 1: W-word weak-LL and SC using 1-word LL, VL, and SC. W-word VL is implemented by validating X.

are eventually helped. If enough space is available, all processes can be helped by one process at the same time — we call this *parallel* helping. Otherwise, several "rounds" of helping must be performed, possibly by several processes — we call this *serial* helping. The tradeoff between serial and parallel helping is one of time versus space.

The remainder of this paper is organized as follows. In Section 2, we present implementations of the LL, SC, and VL operations for large variables discussed above. We then present our lock-free and wait-free universal constructions and preliminary performance results in Section 3. We end the paper with concluding remarks in Section 4. Due to space limitations, we defer detailed proofs to the full paper.

2 LL and SC on Large Variables

In this section, we implement LL, VL, and SC operations for a W-word variable V, where $W > 1$, using the standard, one-word LL, VL, and SC operations.[2] We first present an implementation that supports only the weak-LL operation described in the previous section. We then present an implementation that supports a LL operation with the usual semantics. In the latter implementation, LL is guaranteed to return a "correct" value of V, even if a subsequent SC operation will fail. Unfortunately, this guarantee comes at the cost of higher space overhead and a more complicated implementation. In many applications, however, the weak-LL operation suffices. In particular, in most lock-free and wait-free universal constructions (including ours), LL and SC are used in pairs in such a way that if a SC fails, then none of the computation since the preceding LL has any effect on the object. By using weak-LL, we can avoid such unnecessary computation.

2.1 Weak-LL, VL, and SC Operations for Large Variables

We begin by describing the implementation of weak-LL, VL, and SC shown in Figure 1.[3] The *Long_Weak_LL* and *Long_SC* procedures implement weak-LL and SC operations on a W-word variable V. Values of V are stored in "buffers", and a shared variable X indicates which buffer contains the "current" value of V. The current value is the value written

[2] We assume that the SC operation does not fail spuriously. As shown in [1], a SC operation that does not fail spuriously can be efficiently implemented using LL and a SC operation that might fail spuriously.

[3] Private variables in all figures are assumed to retain their values between procedure calls.

to V by the most recent successful SC operation, or the initial value of V if there is no preceding successful SC. The VL operation for V is implemented by simply validating X.

A SC operation on V is achieved by writing the W-word variable to be stored into a buffer, and by then using a one-word SC operation on X to make that buffer current. To ensure that a SC operation does not overwrite the contents of the current buffer, the SC operations of each process p alternate between two buffers, $BUF[p, 0]$ and $BUF[p, 1]$.

A process p performs a weak-LL operation on V in three steps: first, it executes a one-word LL operation on X to determine which buffer contains the current value of V; second, it reads the contents of that buffer; third, it performs a VL on X to check whether that buffer is still current. If the VL succeeds, then the buffer was not modified during p's read, and the value read by p from that buffer can be safely returned. If the VL fails, then the weak-LL rereads X in order to determine the ID of the last process to perform a successful SC; this process ID is then returned. We call the process whose ID is returned a *witness* of the failed weak-LL. As we will see in Section 3.2, the witness of a failed weak-LL can provide useful state information that held "during" the execution of that weak-LL. Note that if the VL of line 3 fails, then the buffer read by p is no longer current, and hence a subsequent SC by p will fail. This implementation yields the following result.

Theorem 1: Weak-LL, VL, and SC operations for a W-word variable can be implemented using LL, VL, and SC operations for a one-word variable with time complexity $O(W)$, $O(1)$, and $O(W)$, respectively, and space complexity $O(NW)$. □

2.2 LL, VL, and SC Operations for Large Variables

We now show how to implement LL and SC with the "usual" semantics. Although the weak-LL operation implemented above is sufficient for our constructions, other uses of "large" LL and SC might require the LL operation to always return a correct value from V. This is complicated by the fact that all W words of V cannot be accessed atomically.

Our implementation of LL, VL, and SC operations for a W-word variable V is shown in Figure 2. Like the previous implementation, this one employs a shared variable X, along with a set of buffers. Also, a shared array A of "tags" is used for buffer management.

Buffer management differs from that described in the previous subsection in several respects. First, each process p now has $4N + 2$ buffers, $BUF[p, 0]$ to $BUF[p, 4N + 1]$, instead of just two. Another difference is that each buffer now contains more information, specifically an old value of V, a new value of V, and two control bits. The control bits are used to detect concurrent read/write conflicts. These bits, together with the tags in array A, are employed to ensure that each LL returns a correct value, despite any interference.

Figure 2 shows two procedures, *Long_LL* and *Long_SC*, which implement LL and SC operations on V, respectively. As before, a VL on V is performed by simply validating X. The *Long_LL* procedure is similar to the *Long_Weak_LL* procedure, except that, in the event that the VL of X fails, more work is required in order to determine a correct return value. The buffer management scheme employed guarantees the following two properties.

(i) A buffer cannot be modified more than once while some process reads that buffer.

(ii) If a process does concurrently read a buffer while it is being written, then that process obtains a correct value either from the *old* field or from the *new* field of that buffer.

In the full paper, we prove both properties formally. We now describe the implementation shown in Figure 2 in more detail, paying particular attention to (i) and (ii).

In describing the *Long_LL* procedure, we focus on the code that is executed in the event that the VL of X fails, because it is this code that distinguishes the *Long_LL* from the *Long_Weak_LL* procedure of the previous subsection. If a process p executes the *Long_LL*

type *buftype* = **record** b, c: **boolean**; *new*, *old*: **array**$[0..W-1]$ **of** *wordtype* **end**;
tagtype = **record** *pid*: $0..N-1$; *tag*: $0..4N+1$ **end**

shared var X: *tagtype*; BUF: **array**$[0..N-1, 0..4N+1]$ **of** *buftype*; A: **array**$[0..N-1]$ **of** *tagtype*
initially $X = (0,0) \wedge BUF[0,0].b = BUF[0,0].c \wedge BUF[0,0].new =$ initial value of L

private var *val1*, *val2*: **array**$[0..W-1]$ **of** *wordtype*; *curr*, *diff*: *tagtype*; i, j: $0..W-1$; *bit*: **boolean**
initially $j = 1$ and tag 0 is the "last tag sucessfully SC'd"

proc *Long_LL*() **returns array**$[0..W-1]$ **of** *wordtype*

1: $curr := LL(X)$;
for $i := 0$ **to** $W-1$ **do**
2: $val1[i] := BUF[curr.pid, curr.tag].new[i]$
od;
3: **if** $VL(X)$ **then return** *val1*
else
4: $curr := X$;
5: $A[p] := curr$;
for $i := 0$ **to** $W-1$ **do**
6: $val1 := BUF[curr.pid, curr.tag].new[i]$
od;
7: $bit := BUF[curr.pid, curr.tag].b$;
for $i := 0$ **to** $W-1$ **do**
8: $val2[i] := BUF[curr.pid, curr.tag].old[i]$
od;
9: **if** $BUF[curr.pid, curr.tag].c = bit$ **then return** *val2* **else return** *val1*
fi fi

proc *Long_SC*(*newval*: **array**$[0..W-1]$ **of** *wordtype*)

10: **read** $A[j]$;
$j := (j+1)$ **mod** N **fi**;
11: **select** $diff : diff \notin (\{$last N tags read$\} \cup$
$\{$last N tags selected$\} \cup$
$\{$last tag successfully SC'd$\})$;
12: **if** $\neg VL(X)$ **then return** *false* **fi**;
13: $bit := \neg BUF[p, diff].c$;
14: $BUF[p, diff].c := bit$;
for $i := 0$ **to** $W-1$ **do**
15: $BUF[p, diff].old[i] := val1[i]$
od;
16: $BUF[p, diff].b := bit$;
for $i := 0$ **to** $W-1$ **do**
17: $BUF[p, diff].new[i] := newval[i]$
od;
18: **return** $SC(X, (p, diff))$

Figure 2: W-word LL and SC using 1-word LL, VL, and SC. W-word VL is trivially implemented by validating X.

procedure and its VL of X fails, then p might have read a corrupt value from the buffer due to a concurrent write. In order to obtain a correct return value, p reads X again to ascertain the current buffer, and then reads the entire contents of that buffer: *new*, b, *old*, and c. The fields within a buffer are written in the reverse of the order in which they are read in the *Long_LL* procedure. Thus, by property (i), p's read can "cross over" at most one concurrent write by another process. By comparing the values it reads from the b and c fields, p can determine whether the crossing point (if any) occurred while p read the *old* field or the *new* field. Based on this comparison, p can choose a correct return value. This is the essence of the formal proof required to establish property (ii) above.

In describing the *Long_SC* procedure, we focus on the buffer selection mechanism — once a buffer has been selected, this procedure simply updates the *old*, *new*, b, and c fields of that buffer as explained above. The primary purpose of the buffer selection mechanism is to ensure that property (i) holds. Each time a process p executes *Long_SC*, it reads the tag value written to $A[r]$ by some process r (line 10). The tag values are read from the processes in turn, so after N SC operations on V, p has read a tag from each process. Process p selects a buffer for its SC by choosing a new tag (line 11). The new tag is selected to differ from the last N tags read by p from A, to differ from the last N tags selected by p, and to differ from the last tag used in a successful SC by p. The last of these three conditions ensures that p does not overwrite the current buffer, and the first two conditions ensure that property (i) holds. We explain below how tags are selected. First, however, we explain why the selection mechanism ensures property (i).

Observe that, if process q's VL of X (line 3) fails, then before reading from one of p's

```
proc Read_Tag(v)
    if v ∈ Read_Q then
        delete(Read_Q, v);
        enqueue(Read_Q, v)
    else
        enqueue(Read_Q, v);
        delete(Select_Q, v);
        y := dequeue(Read_Q);
        if y ∉ Last_Q then
            enqueue(Select_Q, y)
    fi fi
```

```
proc Store_Tag(v)
    delete(Select_Q, v);
    enqueue(Last_Q, v);
    y := dequeue(Last_Q);
    if y ∉ Read_Q then
        enqueue(Select_Q, y)
    fi
```

```
proc Select_Tag()
        returns 0..4N + 1
    y := dequeue(Select_Q);
    enqueue(Select_Q, y);
    return y
```

Figure 3: Pseudo-code implementations of operations on tag queues.

buffers $BUF[p, v]$ (lines 6 to 9), q writes (p, v) to $A[q]$ (line 5). If p selects and modifies $BUF[p, v]$ while process q is reading $BUF[p, v]$, then p does not select $BUF[p, v]$ again for any of its next N SC operations. Thus, before p selects $BUF[p, v]$ again, p reads $A[q]$ (line 10). As long as (p, v) remains in $A[q]$, it will be among the last N tags read by p, and hence p will not select $BUF[p, v]$ to be modified. Therefore, property (i) holds.

We conclude this subsection by describing how the tag selection in line 11 can be efficiently implemented. To accomplished this, each process maintains three local queues — *Read*, *Last*, and *Select*. The *Read* queue records the last N tags read and the *Last* queue records the last tag successfully written (using SC) to X. All other tags reside in the *Select* queue, from which new tags are selected.

The tag queues are maintained by means of the *Read_Tag*, *Store_Tag*, and *Select_Tag* procedures shown in Figure 3. In these procedures, *enqueue* and *dequeue* denote the normal queue operations, $delete(Q, v)$ removes tag v from Q (and does not modify Q if v is not in Q), and $x \in Q$ holds iff tag x is in queue Q.

Process p selects a tag (line 11 of Figure 2) by calling *Select_Tag*. *Select_Tag* moves the front tag in p's *Select* queue to the back, and returns that tag. If that tag is subsequently written to X by a successful SC operation (line 18), then p calls *Store_Tag* to move the tag from the *Select* queue to the *Last* queue. The tag that was previously in the *Last* queue is removed and, if it is not in the *Read* queue, is returned to the *Select* queue.

When process p reads a tag (p, v) (line 10), it calls *Read_Tag* to record that this tag was read. If (p, v) is already in the *Read* queue, then *Read_Tag* simply moves (p, v) to the end of the *Read* queue. If (p, v) is not already in the *Read* queue, then it is enqueued into the *Read* queue and removed from the *Select* queue, if necessary. Finally, the tag at the front of the *Read* queue is removed because it is no longer one of the last N tags read. If that tag is also not the last tag written to X, then it is returned to the *Select* queue.

The *Read* queue always contains the last N tags read, and the *Last* queue always contains the last tag successfully written to X. Thus, the tag selected by *Select_Tag* is certainly not the last tag successfully written to X, nor is it among the last N tags read. In the full paper, we show that maintaining a total of $4N + 2$ tags ensures that the tag selected is also not one of the last N tags selected, as required.

By maintaining a static index table that allows each tag to be located in constant time, and by representing the queues as doubly-linked lists, all of the queue operations described above can be implemented in constant time. Thus, we have the following result.

Theorem 2: LL, VL, and SC operations for a W-word variable can be implemented using LL, VL, and SC operations for a one-word variable with time complexity $O(W)$, $O(1)$, and $O(W)$, respectively, and space complexity $O(N^2W)$. □

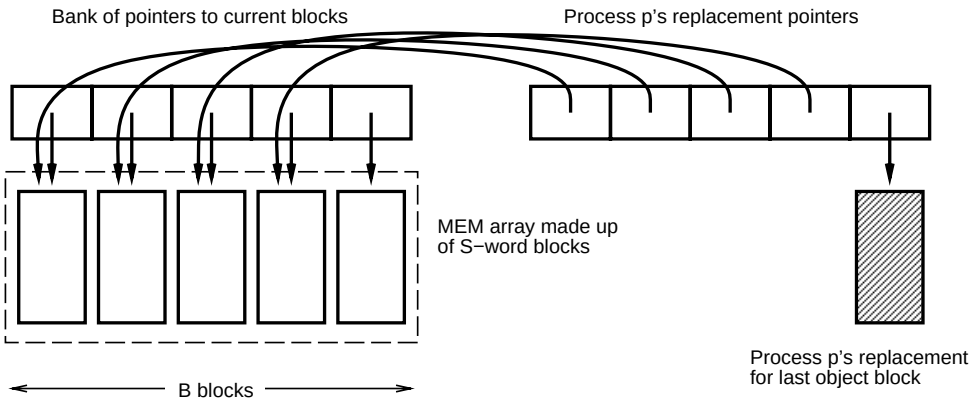

Figure 4: Implementation of the *MEM* array for large object constructions.

3 Large Object Constructions

In this section, we present our lock-free and wait-free universal constructions for large objects. We begin with a brief overview of previous constructions due to Herlihy [4].

Herlihy presented lock-free and wait-free universal constructions for "small" objects as well as a lock-free construction for "large" objects [4]. As described in Section 1, an operation in Herlihy's small-object constructions copies the entire object, which can be a severe disadvantage for large objects. In Herlihy's large-object construction, the implemented object is fragmented into blocks, which are linked by pointers. With this modification, the amount of copying performed by an operation can often be reduced by copying only those blocks that are affected by the operation. However, because of this fragmentation, a significant amount of creative work on the part of the sequential object designer is often required before the advantages of Herlihy's large-object construction can be realized. Also, this approach provides no advantage for common objects such as the queue described in Section 1. Finally, Herlihy did not present a wait-free construction for large objects. Our lock-free and wait-free universal constructions for large objects are designed to overcome all of these problems. These constructions are described next in Sections 3.1 and 3.2, respectively. In Section 3.3, we present performance results comparing our constructions to Herlihy's.

3.1 Lock-Free Universal Construction for Large Objects

Our lock-free construction is shown in Figure 5. In this construction, the implemented object is stored in an array. Unlike Herlihy's small-object constructions, the array is not actually stored in contiguous locations of shared memory. Instead, we provide the illusion of a contiguous array, which is in fact partitioned into blocks. An operation replaces only the blocks it modifies, and thus avoids copying the whole object. Before describing the code in Figure 5, we first explain how the illusion of a contiguous array is provided.

Figure 4 shows an array MEM, which is divided into B blocks of S words each. Memory words $MEM[0]$ to $MEM[S-1]$ are stored in the first block, words $MEM[S]$ to $MEM[2S-1]$ are stored in the second block, and so on. A bank of pointers, one to each block of the array, is maintained in order to record which blocks are currently part of the array. In order to change the contents of the array, an operation makes a copy of each block to be changed, and then attempts to update the bank of pointers by installing new

```
type blktype = array[0..S − 1] of wordtype

shared var   BANK: array[0..B − 1] of 0..B + NT − 1;          /* Bank of pointers to array blocks */
             BLK: array[0..B + NT − 1] of blktype              /* Array and copy blocks */
initially (∀k : 0 ≤ k < B :: BANK[k] = NT + k ∧ BLK[NT + k] = (kth block of initial value))

private var  oldlst, copy: array[0..T − 1] of 0..B + NT − 1;  ptrs: array[0..B − 1] of 0..B + NT − 1;
             dirty: array[0..B − 1] of boolean;  dirtycnt: 0..T;  i, blkidx: 0..B − 1;
             blk: 0..B + NT − 1;  ret: objrettype
initially (∀k : 0 ≤ k < T :: copy[k] = pT + k)

proc Read(addr: 0..BS − 1) returns wordtype
    return BLK[ptrs[addr div S]][addr mod S]

proc Write(addr: 0..BS − 1; val: wordtype)
    blkidx := addr div S;                                     /* Compute block index from address */
    if ¬dirty[blkidx] then                                    /* Haven't changed this block before */
        dirty[blkidx] := true;                                /* Record that block is changed */
        memcpy(BLK[copy[dirtycnt]], BLK[ptrs[blkidx]], sizeof(blktype)); /* Copy old block to new */
        oldlst[dirtycnt], ptrs[blkidx], dirtycnt := ptrs[blkidx], copy[dirtycnt], dirtycnt + 1
    fi;                                                       /* Install new block, record old block, prepare for next one */
    BLK[ptrs[blkidx]][addr mod S] := val                      /* Write new value */

proc LF_Op(op: optype; pars: paramtype)
    while true do                                             /* Loop until operation succeeds */
1:      if Long_Weak_LL(BANK, ptrs) = N then                  /* Load object pointer */
            for i := 0 to B − 1 do dirty[i] := false od;  dirtycnt := 0;   /* No blocks copied yet */
2:          ret := op(pars);                                  /* Perform operation on object */
3:          if dirtycnt = 0 ∧ Long_VL(BANK) then return ret fi;   /* Avoid unnecessary SC */
4:          if Long_SC(BANK, ptrs) then                       /* Operation is successful, reclaim old blocks */
                for i := 0 to dirtycnt − 1 do copy[i] := oldlst[i] od;
                return ret
        fi  fi
    od
```

Figure 5: Lock-free implementation for a large object.

pointers for the changed blocks; the other pointers are left unchanged. This is achieved by using the weak-LL and SC operations for large variables presented in Section 2.1.[4] In Figure 4, process p is preparing to modify a word in the last block, but no others. Thus, the bank of pointers to be written by p is the same as the current bank, except that the last pointer points to p's new last block.

When an operation by process p accesses a word in the array, say $MEM[x]$, the block that currently contains $MEM[x]$ must be identified. If p's operation modifies $MEM[x]$, then p must replace that block. In order to hide the details of identifying blocks and of replacing modified blocks, some address translation and record-keeping is necessary. This work is performed by special *Read* and *Write* procedures, which are called by the sequential operation in order to read or write the *MEM* array. As a result, our constructions are not completely transparent to the sequential object designer. For example, instead of writing "$MEM[1] := MEM[10]$", the designer would write "$Write(1, Read(10))$". However, as discussed in Section 4, a preprocessor could be used to provide complete transparency.

We now turn our attention to the code of Figure 5. In this figure, *BANK* is a B-word shared variable, which is treated as an array of B pointers (actually indices into the *BLK* array), each of which points to a block of S words. Together, the B blocks pointed to by *BANK* make up the implemented array *MEM*. We assume an upper bound T on the number of blocks modified by any operation. Therefore, in addition to the B

[4] An extra parameter has been added to the procedures of Section 2.1 to explicitly indicate which shared variable is updated.

blocks required for the object, T "copy blocks" are needed per process, giving a total of $B + NT$ blocks. These blocks are stored in the *BLK* array. Although blocks $BLK[NT]$ to $BLK[NT + B - 1]$ are the initial array blocks, and $BLK[pT]$ to $BLK[(p+1)T - 1]$ are process p's initial copy blocks, the roles of these blocks are not fixed. In particular, if p replaces a set of array blocks with some of its copy blocks as the result of a successful SC, then p reclaims the replaced array blocks as copy blocks. Thus, the copy blocks of one process may become blocks of the array, and later become copy blocks of another process.

Process p performs a lock-free operation by calling the *LF_Op* procedure. The loop in the *LF_Op* procedure repeats until the SC at line 3 succeeds. In each iteration, process p first reads *BANK* into a local variable *ptrs* using a B-word weak-LL. Recall from Section 2.1 that the weak-LL can return a process identifier from $\{0, ..., N-1\}$ if the following SC is guaranteed to fail. In this case, there is no point in attempting to apply p's operation, so the loop is restarted. Otherwise, p records in its *dirty* array that no block has yet been modified by its operation, and initializes the *dirtycnt* counter to zero.

Next, p calls the *op* procedure provided as a parameter to *LF_Op*. The *op* procedure performs the sequential operation by reading and writing the elements of the *MEM* array. This reading and writing is performed by invoking the *Read* and *Write* procedures shown in Figure 5. The *Read* procedure simply computes which block currently contains the word to be accessed, and returns the value from the appropriate offset within that block. The *Write* procedure performs a write to a word of *MEM* by computing the index *blkidx* of the block containing the word to be written. If it has not already done so, the *Write* procedure then records that the block is "dirty" (i.e., has been modified) and copies the contents of the old block to one of p's copy blocks. Then, the copy block is linked into p's *ptrs* array, making that block part of p's version of the *MEM* array, and the displaced old block is recorded in *oldlst* for possible reclaiming later. Finally, the appropriate word of the new block is modified to contain the value passed to the *Write* procedure.

If *BANK* is not modified by another process after p's weak-LL, then the object contained in p's version of the *MEM* array (pointed to by p's *ptrs* array) is the correct result of applying p's operation. Therefore, p's SC successfully installs a copy of the object with p's operation applied to it. After the SC, p reclaims the displaced blocks (recorded in *oldlst*) to replace the copy blocks it used in performing its operation. On the other hand, if another process *does* modify *BANK* between p's weak-LL and SC, then p's SC fails. In this case, some other process completes an operation, so the implementation is lock-free.

Before concluding this subsection, one further complication bears mentioning. If the *BANK* variable is modified by another process while p's sequential operation is being executed, then it is possible for p to read inconsistent values from the *MEM* array. Observe that this does not result in p installing a corrupt version of the object, because p's subsequent SC fails. However, there is a risk that p's sequential operation might cause an error, such as a division by zero or a range error, because it reads an inconsistent state of the object. This problem can be solved by ensuring that, if *BANK* is invalidated, control returns directly from the *Read* procedure to the *LF_Op* procedure, without returning to the sequential operation. The Unix `longjmp` command can be used for this purpose. The details are omitted from Figure 5. In the full paper, we prove the following.

Theorem 3: Suppose a sequential object *OBJ* can be implemented in an array of B S-word blocks such that any operation modifies at most T blocks and has worst-case time complexity C. Then, *OBJ* can be implemented in a lock-free manner with space overhead[5] $O(NB + NTS)$ and contention-free time complexity $O(B + C + TS)$. □

It is interesting to compare these complexity figures to those of Herlihy's lock-free

[5] By *space overhead*, we mean space complexity beyond that required for the sequential object.

construction. Consider the implementation of a queue. By storing head and tail "pointers" (actually, array indices, not pointers) in a designated block, an enqueue or dequeue can be performed in our construction by copying only two blocks: the block containing the head or tail pointer to update, and the block containing the array slot pointed to by that pointer. Space overhead in this case is $O(NB + NS)$, which should be small when compared to $O(BS)$, the size of the queue. Contention-free time complexity is $O(B + C + S)$, which is only $O(B + S)$ greater than the time for a sequential enqueue or dequeue. In contrast, as mentioned in Section 1, each process in Herlihy's construction must actually copy the entire queue, even when using his large-object techniques. Thus, space overhead is at least N times the worst-case queue length, i.e., $\Omega(NBS)$. Also, contention-free time complexity is $\Omega(BS + C)$, since $\Omega(BS)$ time is required to copy the entire queue in the worst case.

When implementing a balanced tree, both constructions require space overhead of $O(N \log(BS))$ for local blocks. However, we pay a logarithmic time cost only when performing an operation whose sequential counterpart modifies a logarithmic number of array slots. In contrast, Herlihy's construction entails a logarithmic time cost for copying for almost every operation — whenever some block is modified, a chain of block pointers must be updated from that block to the block containing the root of the tree.

3.2 Wait-Free Construction for Large Objects

Our wait-free construction for large objects is shown in Figure 6. As in the lock-free construction presented in the previous subsection, this construction uses the *Read* and *Write* procedures in Figure 5 to provide the illusion of a contiguous array. The principal difference between our lock-free and wait-free constructions is that processes in the wait-free construction "help" each other in order to ensure that each operation by each process is eventually completed. To enable each process to perform the operation of at least one other process together with its own, each process p now has $M \geq 2T$ private copy blocks. (Recall that T is the maximum number of blocks modified by a single operation.)

The helping mechanism used in our wait-free, large-object construction is similar to that used in Herlihy's wait-free, small-object construction in several respects. To enable processes to perform each others' operations, each process q begins by "announcing" its operation and parameters in $ANC[q]$ (line 11 in Figure 6). Also, each process stores sufficient information with the object to allow a helped process to detect that its operation was completed and to determine the return value of that operation. This information also ensures that the operation helped is not subsequently reapplied.

There are also several differences between our helping mechanism and Herlihy's. First, in Herlihy's construction, each time a process performs an operation, it also performs the pending operations of all other processes. However, in our construction, the restricted amount of private copy space might prevent a process from simultaneously performing the pending operations of all other processes. Therefore, in our construction, each process helps only as many other processes as it can with each operation. In order to ensure that each process is eventually helped, a *help* counter is added to the shared variable *BANK* used in our lock-free construction. The *help* field indicates which process should be helped next. Each time process p performs an operation, p helps as many processes as possible starting from the process stored in the *help* field. This is achieved by helping processes until too few private copy blocks remain to accommodate another operation (lines 22 to 24). (Recall that the *Write* procedure in Figure 5 increments *dirtycnt* whenever a new block is modified.) Process p updates the *help* field so that the next process to successfully perform an operation starts helping where p stops.

Our helping mechanism also differs from Herlihy's in the way a process detects the completion of its operation. In Herlihy's construction, completion is detected by means

```
type anctype = record op: optype; pars: paramtype; bit: boolean end;
     retblktype = array[0..N − 1] of record val: objrettype; applied, copied: boolean end
     blktype = array[0..S − 1] of wordtype;
     banktype = record blks: array[0..B − 1] of 0..B + NM − 1; help: 0..N − 1; ret: 0..N end

shared var  BANK: banktype;  BLK: array[0..B + NM − 1] of blktype;
            ANC: array[0..N − 1] of anctype;                          /* Announce array */
            RET: array[0..N] of retblktype;                  /* Blocks for operation return values */
            LAST: array[0..N − 1] of 0..N          /* Last RET block updated by each process */
initially BANK.ret = N ∧ (∀p :: ANC[p].bit = RET[N][p].applied = RET[N][p].copied) ∧
BANK.help = 0 ∧ (∀k : 0 ≤ k < B :: BANK.blks[k] = NM + k ∧ BLK[NM + k] = (kth initial block))

private var oldlst, copy: array[0..M − 1] of 0..B + NM − 1;  b, tmp, rb, oldrb: 0..N;  ptrs: banktype;
      match, done, bit, a, loop: boolean;  applyop: optype;  applypars: paramtype;  j, try: 0..N − 1;
      m: 0..M − 1;  dirty: array[0..B − 1] of boolean;  dirtycnt: 0..M;  i: 0..B − 1
initially (∀k : 0 ≤ k < M :: copy[k] = pM + k) ∧ rb = p ∧ ¬bit

proc Apply(q: 0..N − 1)
1:  match := ANC[q].bit;
2:  if RET[rb][q].applied ≠ match then
3:      applyop := ANC[q].op;
4:      applypars := ANC[q].pars;
5:      RET[rb][q].val := applyop(applypars);
6:      RET[rb][q].applied := match
    fi

proc Return_Block() returns 0..N
7:  tmp := Long_Weak_LL(BANK, ptrs);
8:  if tmp ≠ N then
9:      return LAST[tmp]
    else
10:     return ptrs.ret
    fi

proc WF_Op(op: optype; pars: paramtype)
11: ANC[p], bit := (op, pars, ¬bit), ¬bit;                          /* Announce operation */
12: b, done := Return_Block(), false;
13: while ¬done ∧ RET[b][p].copied ≠ bit do /* Loop until update succeeds or operation is helped */
14:     if Long_Weak_LL(BANK, ptrs) = N then                          /* Load object pointers */
15:         for i := 0 to B − 1 do dirty[i] := false od;  dirtycnt := 0;     /* No blocks modified yet */
16:         oldrb, ptrs.ret := ptrs.ret, rb;            /* Record old return block and install new one */
17:         memcpy(RET[rb], RET[oldrb], sizeof(retblktype)); /* Make private copy of return block */
18:         if Long_VL(BANK) then                                  /* Check if Long_SC will fail */
                for j := 0 to N − 1 do                             /* Record applied operations */
19:                 a := RET[rb][j].applied;
20:                 RET[rb][j].copied := a
                od;
21:             Apply(p);  try, loop := ptrs.help, false;                  /* Apply own operation */
22:             while dirtycnt + T ≤ M ∧ ¬loop do /* Help processes while sufficient space remains */
23:                 Apply(try);
24:                 try := try + 1 mod N;  if try = ptrs.help then loop := true fi
                od;
25:             LAST[p], ptrs.help := rb, try;          /* Relay which return block was modified */
26:             if Long_SC(BANK, ptrs) then          /* Operation is successful, reclaim old blocks */
27:                 for m := 0 to dirtycnt − 1 do copy[m] := oldlst[m] od;
28:                 RET[rb][p].copied, rb, done := bit, oldrb, true /* Prepare copied bit for next time */
                fi
            fi
        fi;
29:     b := Return_Block()                                  /* Get current or recent return block */
    od;
30: return RET[b][p].val                                        /* Get return value of operation */
```

Figure 6: Wait-free implementation for a large object.

of a collection of toggle bits, one for each process, that are stored with the current version of the object. Before attempting to apply its operation, each process p first "announces" a new toggle bit value. When another process helps p, it copies this bit value into the current version of the object. To detect the completion of its operation, p tests whether the bit value stored for it in the current version of the object matches the bit value it previously announced; to access the current version of the object, p first reads the shared object pointer, and then reads the buffer pointed to by that pointer. In order to avoid a race condition that can result in an operation returning an incorrect value, Herlihy's construction requires this sequence of reads to be performed twice. This race condition arises when p attempts to access the current buffer, and during p's access, another process subsequently reclaims that buffer and privately updates it. By dereferencing the object pointer and checking its toggle bit a second time, p can ensure that if the first buffer it accessed has been reclaimed, then p's operation has already been applied. This is because the process that reclaimed the buffer helped all other processes with its operation, and therefore ensured that p's operation was applied. Because our construction does not guarantee that each process helps all other processes at once, p might have to reread the shared object pointer and read its toggle bit many times to ensure that its operation has been applied. We therefore use a different mechanism, explained below, for determining whether an operation has been applied.

To enable a process to detect that its operation has been applied, and to determine the return value of the operation, we use a set of "return" blocks. There are $N+1$ return blocks $RET[0]$ to $RET[N]$; at any time, one of these blocks is "current" (as indicated by a new *ret* field in the $BANK$ variable) and each process "owns" one of the other return blocks. The current return block contains, for each process q, the return value of q's most recent operation, along with two bits: *applied* and *copied*. These bits are used by q to detect when its operation has been completed. Roughly speaking, the *applied* bit indicates that q's operation has been applied to the object and the *copied* bit indicates that another operation has been completed since q's operation was applied. The interpretation of these bits is determined by $ANC[q].bit$. For example, q's operation has been applied iff q's *applied* bit in the current return block equals $ANC[q].bit$.

To see why two bits are needed to detect whether q's operation is complete, consider the scenario in Figure 7. In this figure, process p performs two operations. In the first, p's SC is successful, and p replaces $RET[5]$ with $RET[3]$ as the current return block at line 26. During p's first operation, q starts an operation. However, q starts this operation too late to be helped by p. Before p's execution of line 26, q reads $BANK$ in line 7 and determines that $RET[5]$ is the current return block. Now, p starts a second operation. Because p previously replaced $RET[5]$ as the current return block, $RET[5]$ is now p's private copy, so p's second operation uses $RET[5]$ to record the operations it helps. When p executes line 6, it changes q's *applied* bit to indicate that it has applied q's operation. Note that, at this stage, q's operation has only been applied to p's private object copy, and p has not yet performed its SC. However, if q reads the *applied* bit of $RET[5]$ (which it previously determined to be the current RET block) at line 13, then q incorrectly concludes that its operation has been applied to the object, and terminates prematurely.

It is similarly possible for q to detect that its *copied* bit in some return block $RET[b]$ equals $ANC[q].bit$ before the SC (if any) that makes $RET[b]$ current. However, because q's *copied* bit is updated only *after* its *applied* bit has been successfully installed as part of the current return block, it follows that some process must have previously applied q's operation. Thus, q terminates correctly in this case (see line 13).

It remains to describe how process q determines which return block contains the current state of q's operation. It is not sufficient for q to perform a weak-LL on $BANK$ and read the *ret* field, because the weak-LL is not guaranteed to return a value of $BANK$ if a successful

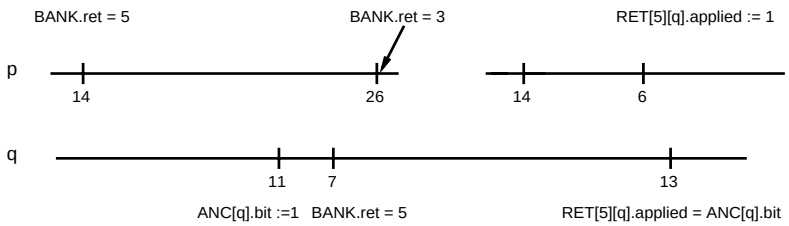

Figure 7: Process q prematurely detects that its *applied* bit equal $ANC[q].bit$.

SC operation interferes. In this case, the weak-LL returns the ID of a "witness" process that performs a successful SC on $BANK$ during the weak-LL operation. In preparation for this possibility, process p records the return block it is using in $LAST[p]$ (line 25) before attempting to make that block current (line 26). When q detects interference from a successful SC, q uses the $LAST$ entry of the witness process to determine which return block to read. The $LAST$ entry contains the index of a return block that was current during q's weak-LL operation. If that block is subsequently written after being current, then it is a copy of a more recent current return block, so its contents are still valid. Our wait-free construction gives rise to the following result.

Theorem 4: Suppose a sequential object OBJ whose return values are at most R words can be implemented in an array of B S-word blocks such that any operation modifies at most T blocks and has worst-case time complexity C. Then, for any $M \geq 2T$, OBJ can be implemented in a wait-free manner with space overhead $O(N(NR + MS + B))$ and worst-case time complexity $O(\lceil N/\min(N, \lfloor M/T \rfloor)\rceil(B + N(R + C) + MS))$.[6] □

3.3 Performance Comparison

In this subsection, we describe the results of preliminary experiments that compare the performance of Herlihy's lock-free construction for large objects to our two constructions on a 32-processor KSR-1 multiprocessor.

The results of one set of experiments are shown in Figure 8. In these experiments, LL and SC primitives were implemented using native KSR locks. Each of 16 processors performed 1000 enqueues and 1000 dequeues on a shared queue. For testing our constructions, we chose B (the number of blocks) and S (the size of each block) to be approximately the square root of the total object size. Also, we chose $T = 2$ because each queue operation accesses only two words. For the wait-free construction, we chose $M = 4$. This is sufficient to guarantee that each process can help at least one other operation. In fact, because two consecutive enqueue (or dequeue) operations usually access the same block, choosing $M = 4$ is sufficient to ensure that a process often helps all other processes each time it performs an operation. These choices for M and T result in very low space overhead compared to that required by Herlihy's construction.

As expected, both our lock-free and wait-free constructions significantly outperform Herlihy's construction as the queue size grows. This is because an operation in Herlihy's construction copies the entire object, while ours copy only small parts of the object.

It is interesting to note that our wait-free construction outperforms our lock-free one.

[6] It can be shown that each successful operation is guaranteed to advance the help pointer by $\min(N, \lfloor M/T \rfloor)$. Thus, if process p's SC fails $\lceil N/\min(N, \lfloor M/T \rfloor)\rceil$ times, then p's operation is helped. When considering these bounds, note that for many objects, R is a small constant. Also, for queues, C and T are constant, and for balanced trees, C and T are logarithmic in the size of the object.

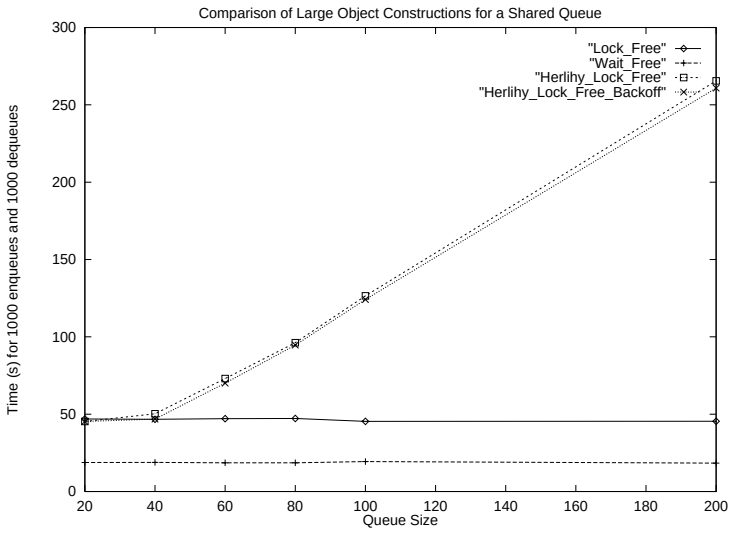

Figure 8: Performance experiments on KSR. $N = 16$, $T = 2$, $M = 4$.

We believe that this is because the cost of recopying blocks in the event that a SC fails dominates the cost of helping. It is also interesting to note that exponential backoff does not significantly improve the performance of Herlihy's lock-free construction. This stands in contrast to Herlihy's experiments on small objects, where exponential backoff played an important role in improving performance. We believe that this is because the performance of Herlihy's large object construction is dominated by copying and not by contention.

We should point out that we have deliberately chosen the queue to show the advantages of our constructions over Herlihy's. In the full paper, we will also present an implementation of a skew heap — the object considered by Herlihy. We expect that our constructions will still outperform Herlihy's, albeit less dramatically, because ours will copy a logarithmic number of blocks only when the sequential operation does; Herlihy's will do so whenever a block near the bottom of the tree is modified.

4 Concluding Remarks

Our constructions improve the space and time efficiency of lock-free and wait-free implementations of large objects. Also, in contrast to similar previous constructions, ours do not require programmers to determine how an object should be fragmented, or how the object should be copied. However, they do require the programmer to use special *Read* and *Write* functions, instead of the assignment statements used in conventional programming. Nonetheless, as demonstrated by Figure 9, the resulting code is very close to that of an ordinary sequential implementation. Our construction could be made completely seamless by providing a compiler or preprocessor that automatically translates assignments to and from *MEM* into calls to the *Read* and *Write* functions.

The applicability of our construction could be further improved by the addition of a dynamic memory allocation mechanism. This would provide a more convenient interface for objects such as balanced trees, which are naturally represented as nodes that are dynamically allocated and released. There are well-known techniques for implementing dynamic memory management in an array. These techniques could be applied directly by the sequential object programmer, or could be provided as a subroutine library. Several issues arise from the design of such a library. First, the dynamic memory allocation pro-

```
int dequeue()
{
 int item;

 if (Read(head) == Read(tail))
   return EMPTY;
 item = Read(Read(head));
 Write(head,(Read(head)+1)%n);
 return item;
}
```

```
int enqueue(item)
     int item;
{
 int newtail;                /* int newtail;            */

 Write(Read(tail),item);     /* MEM[tail] = item;       */
 newtail = (Read(tail)+1)%n; /* newtail = (tail+1) % n; */
if (newtail == Read(head))   /* if (newtail == head)    */
   return FULL;              /*   return FULL;          */
 Write(tail,newtail);        /* tail = newtail;         */
 return SUCCESS;             /* return SUCCESS;         */
}
```

Figure 9: C code used for the queue operations. Comments show "usual" enqueue code.

cedures must modify only a small number of array blocks, so that the advantages of our constructions can be preserved. Second, fragmentation complicates the implementation of *allocate* and *release* procedures. These complications can make the procedures quite inefficient, and can even cause the *allocate* procedure to incorrectly report that insufficient memory is available. Both of these problems are significantly reduced if the size of allocation requests is fixed in advance. For many objects, this restriction is of no consequence. For example, the nodes in a tree are typically all of the same size.

Finally, our constructions do not allow parallel execution of operations, even if the operations access disjoint sets of blocks. We would like to extend our constructions to allow such parallel execution where possible. For example, in our shared queue implementations, an *enqueue* operation might unnecessarily interfere with a *dequeue* operation. In [1], we addressed similar concerns when implementing wait-free operations on multiple objects.

Acknowledgement: We would like to thank Lars Nyland for his help with the performance studies in Section 3.3.

References

[1] J. Anderson and M. Moir, "Universal Constructions for Multi-Object Operations", to appear in the *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[2] G. Barnes, "A Method for Implementing Lock-Free Shared Data Structures", *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 261-270.

[3] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[4] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 5, 1993, pp. 745-770.

[5] A. Israeli and L. Rappoport, "Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives", *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1994, pp. 151-160.

[6] N. Shavit and D. Touitou, "Software Transactional Memory", to appear in the *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995.