Implementing Wait-Free Objects on Priority-Based Systems*

James H. Anderson, Srikanth Ramamurthy, and Rohit Jain
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Wait-free objects are often implemented through the use of a "helping scheme", whereby one process "helps" one or more other processes to complete an operation. This paper presents several new helping schemes that can be generally applied to efficiently implement a variety of different objects on priority-based uniprocessor and multiprocessor systems. Examples of such systems include lock-free multiprocessor kernels and real-time systems. Our helping schemes reduce overhead by exploiting the way in which processes are scheduled in priority-based systems. We illustrate the use of these schemes by presenting wait-free implementations of linked lists and a multi-word compare-and-swap primitive.

1 Introduction

We consider the implementation of wait-free shared objects on multiprogrammed systems in which processes are scheduled for execution based on priority. We assume that processes are scheduled on a per-processor basis and do not migrate between processors during object accesses. Our work extends research reported in a recent paper by Ramamurthy, Moir, and Anderson [12] pertaining to the implementation of wait-free objects in priority-based real-time systems. Although Ramamurthy et al. targeted real-time systems as the main application area for their work, their results actually are applicable to any priority-based system, as long as the following conditions are satisfied: (i) on a given processor, a process p may preempt another process q only if p has higher priority than q; and (ii) a process's priority can change over time, but not during any object access. A good example of a non-real-time system that satisfies these requirements is a multiprocessor kernel. As long as interrupts in such a system are handled in a priority-based manner, and objects are pinned in memory, conditions (i) and (ii) apply (viewing the code to handle a given interrupt as a distinct "process"). Wait-free and lock-free kernel data structures facilitate the design of re-entrant kernels, because their use eliminates the possibility of deadlock resulting from a preempted object access. Examples of lock-free kernels include the Synthesis Kernel of Massalin and Pu [10] and the Cache Kernel of Greenwald and Cheriton [7].

One of the main lessons to be learned from [12] is that many problems that are either difficult or impossible to solve in a wait-free manner in asynchronous systems *can* be solved efficiently in a wait-free manner in priority-based systems. The reason for this is that, in a priority-based system, operations of high-priority processes *automatically* appear atomic to low-priority processes running on the same processor. (This is illustrated quite well by Figure 2, which we consider below.) This fact can be exploited to greatly simplify object implementations.

In [12], several wait-free algorithms are presented for implementing consensus and compare-and-swap (`CAS`) objects in priority-based systems. In this paper, we consider objects that are more sophisticated than these, and present several new algorithmic techniques that can be generally applied in priority-based systems to construct wait-free implementations of objects. We illustrate the use of these techniques by presenting wait-free implementations of linked lists and a multi-word compare-and-swap (`MWCAS`) primitive. Our results are summarized in Figure 1.

In the first part of the paper, we focus our attention on results pertaining to priority-based uniprocessor systems. We begin by presenting a wait-free implementation of `MWCAS`. In this implementation, a W-word `MWCAS` operation is executed in $\Theta(W)$ time, which is asymptotically optimal. The `MWCAS` implementation presented here is based on one presented previously by Anderson and Ramamurthy [4]. Anderson and Ramamurthy's implementation is not strictly faithful to the semantics of `MWCAS`. In particular, it allows a `MWCAS` operation to fail if it is overlapped by another `MWCAS` operation that accesses a common word. For the case in which the overlapping operation itself fails, it may be impossible to correctly linearize the operations in accordance with the semantics of `MWCAS` (see [4] for details).[1] In the implementation presented here, this problem is corrected.

After considering `MWCAS`, we present an implementation of linked lists for priority-based uniprocessor systems. Our list implementation is based on a novel technique, which we call *incremental helping*. This technique is of general utility when implementing wait-free objects on priority-based systems, and is also used in our multiprocessor implementations. The general idea of incremental helping is illustrated in Figure 2. Before beginning an operation, a process must first "announce" its intentions by writing information about its operation into a shared "announce variable". Before a process is allowed to do this, however, it must first help any previously-announced operation (on its processor) to complete execution. This scheme requires only one announce variable per processor. In contrast, previous constructions for asynchronous systems require one announce variable per process [2, 3, 8]. In addition, with incremental helping, each process helps at most one other process, while in helping

*Work supported, in part, by NSF grants CCR 9216421 and CCR 9510156, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323. The first author was also supported by an Alfred P. Sloan Research Fellowship.

[1] For the application considered by Anderson and Ramamurthy, this is not a problem. Their MWCAS algorithm is intended to be used along with a real-time scheduling scheme that ensures that sufficient processor time exists to retry failed operations.

	Uniprocessor Implementations		Multiprocessor Implementations	
	Primitives Used	Worst-Case Time	Primitives Used	Worst-Case Time
`MWCAS`	`CAS`	$\Theta(W)$	`CAS`, `CCAS`	$\Theta(2PW)$
linked lists	`CAS`	$\Theta(2T)$	`CAS`, `CCAS`	$\Theta(2PT)$

Figure 1: Summary of results. P denotes the number of processors in a system. W denotes the number of words accessed by a `MWCAS`. T is the worst-case time for a list operation. The constant 2 is used in the Θ notation to more accurately reflect the cost of helping.

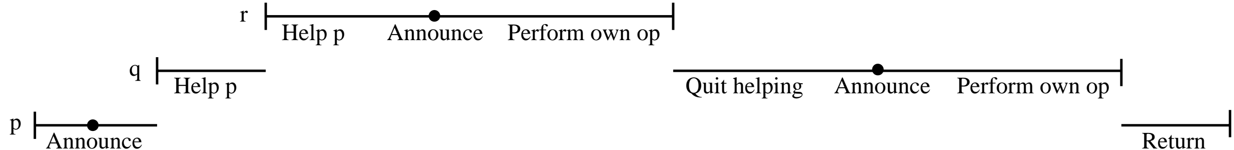

Figure 2: Process p detects no previously-announced process, so it announces its operation. Before p can complete its operation, it is preempted by process q. Process q begins to help p to complete its operation, but before it finishes, it is preempted by process r. Process r detects that p's operation has been announced but is not finished, so it too helps p. It then announces its own operation, executes it, and relinquishes the processor to process q. Process q detects that p's operation is complete, so it announces its own operation, executes it, and relinquishes the processor to p. Process p detects that its operation has been completed, so it returns.

schemes for asynchronous systems, each process helps all other processes in the worst case.

In the second part of the paper, we consider the implementation of wait-free objects on priority-based multiprocessors. Again, we present implementations of both a `MWCAS` primitive and linked lists. As mentioned above, we make use of incremental helping in our multiprocessor implementations. In addition, we use two other new techniques, which we call *cyclic helping* and *priority helping*, respectively. The notion of cyclic helping is adapted from previous work of Anderson and Moir [3]. In this scheme, the processors are thought of as if they were part of a logical ring. Processes are helped through the use of a "help counter", which cycles around the ring. To advance the help counter from processor R to the next processor on the ring, a process must first help the currently-announced process on processor R. In order to perform an operation, a process does the following: it first repeatedly advances the help counter until any pending announced (lower-priority) operation on its own processor has been completed; it then announces its own operation; it then repeatedly advances the help counter until is own operation has been completed. Priority helping is a variation on cyclic helping in which the help counter, instead of being advanced around a logical ring, is always advanced to the processor with highest-priority pending operation.

Our multiprocessor implementations make use of a special instruction, which we call *conditional compare-and-swap* (`CCAS`). `CCAS` is a restriction of the more well-known two-word compare-and-swap (`CAS2`) instruction in which one word is a compare-only version number. This version number is incremented after each wait-free operation and is assumed to not cycle during any single operation.[2] Unfortunately, `CAS2` is directly provided on only a few existing processors (e.g., Motorola 68030 and 68040). Algorithms for implementing `CAS2` *are* known [2, 5, 9], but none are efficient enough to be practically applied. An efficient hardware-based implementation of `CAS2` was recently proposed by Greenwald and Cheriton [7], but no current machines support this implementation. An operating-system-based approach to implementing `CAS2` has been proposed by Bershad [6], but this approach can be problematic to actually implement (see [7] for details). Fortunately, `CCAS` appears to be much easier to implement than `CAS2`. If `CAS` is available, then `CCAS` can be implemented in just a few instructions.

When using cyclic helping to implement an object that is shared across P processors, the time to perform an operation is proportional to $2 \cdot P \cdot T$, where T is the time required for the longest operation. With previous universal constructions, the time to perform an operation in the worst case is $2 \cdot N \cdot T$, where N is the number of processes sharing an object.[3] If P were very large, then our algorithms would not be much better than previous ones. However, our main interest is in real-time systems, and it is much more likely that such systems would be implemented on a workstation-class machine, as opposed to a very large multiprocessor. On multiprocessor workstations, the number of processors is typically small to moderate (probably 16 or fewer processors). Even on a large machine, the likelihood of having an object that is shared across more than a moderate number of processors is probably small. In the latter part of the paper, we present performance results that show that our multiprocessor linked-list implementation is more efficient than alternative implementations on priority-based multiprocessor workstations.

In work on multi-word synchronization primitives like `MWCAS`, Israeli and Rappoport introduced the important notion of *disjoint access parallelism* [9]. For a wait-free implementation to be disjoint access parallel, a process should only help other processes with which it has a transitive conflict.[4] Our multiprocessor `MWCAS` implementation does not have this property. However, while ensuring parallelism in this context is of theoretical importance, *practical* algorithms that are disjoint access parallel have remained elu-

[2] This is reasonable for the kinds of applications targeted by our work. For example, in real-time systems, each task must complete execution by a specified deadline. Unless deadlines are unrealistically large, it would be impossible for a 32- or 64-bit counter to cycle during the execution of one task.

[3] For example, in Herlihy's construction [8], two rounds of helping may be required, with N processes being helped per round.

[4] For instance, if processes p, q, and r concurrently perform MW-CAS operations on words w and x, x and y, and y and z, respectively, then p has a direct conflict with q and transitive conflicts with both q and r.

```
type                                        /* Assume N processes, each MWCAS accesses at most B words */
    wordtype = record val: valtype; cnt: 0..B − 1; valid: boolean; pid: 0..N − 1 end;   /* All of these fields are
            stored in one word; the val field is application dependent; the valid field should be initially true */

    addrlisttype = array[0..B − 1] of pointer to wordtype;              /* Addresses to perform MWCAS on */
    vallisttype = array[0..B − 1] of valtype                            /* List of old and new values for MWCAS */

shared variable
    Status: array[0..N − 1] of integer initially 0;   /* Status of process's latest MWCAS: 0 if pending, 1 if invalid, 2 if valid */
    Save: array[0..N − 1, 0..B − 1] of valtype     /* Used to temporarily save value from a word during a MWCAS on that word */

private variable                                                        /* For process p, where 0 ≤ p < N */
    init, assn: array[0..B − 1] of wordtype;             /* Values initially read and assigned to words by MWCAS */
    i,j: 0..B + 1; retval: boolean; word: wordtype; val: valtype

procedure MWCAS(numwds: 0..B; addr: addrlisttype;
                old, new: vallisttype) returns boolean
1:  Status[p] := 0;
2:  i := 0;
3:  while i < numwds ∧ Status[p] = 0 do
4:    init[i] := *addr[i];
5:    if init[i].valid ∨ Status[init[i].pid] = 2 then
6:      val := init[i].val
      else
7:      val := Save[init[i].pid, init[i].cnt]
      fi;
8:    Save[p, i] := val;
9:    if old[i] ≠ val then
10:     Status[p] := 1
      else
11:     assn[i] := (new[i], i, false, p);
12:     if ¬CAS(addr[i], init[i], assn[i]) then
13:       Status[p] := 1
        fi;
14:     i := i + 1
      fi
    od;

/* MWCAS continued */
15: retval := CAS(&Status[p], 0, 2);
16: for j := 0 to i − 1 do
17:   if old[j] ≠ new[j] ∧ retval then
18:     CAS(addr[j], assn[j], (new[j], 0, true, p));
19:     if ¬init[j].valid then CAS(&Status[init[j].pid], 0, 1) fi
20:   else if ¬CAS(addr[j], assn[j], init[j]) then
21:     if ¬init[j].valid then CAS(&Status[init[j].pid], 0, 1) fi
      fi
    od;
22: return(retval)

procedure Read(addr: pointer to wordtype) returns valtype
23: word := *addr;
24: if word.valid ∨ Status[word.pid] = 2 then
25:   return(word.val)
    else
26:   return(Save[word.pid, word.cnt])
    fi
```

Figure 3: Wait-free implementation of MWCAS for priority-based uniprocessors.

sive. Indeed, if existing algorithms are any indication [2, 5, 9], disjoint access parallelism, while improving best-case complexity, results in enormous worst-case complexity. In our multiprocessor MWCAS implementation, a W-word MWCAS operation on P processors requires $O(PW)$ time. When considering multiprocessor workstations, it is reasonable to consider P to be a constant. Under this assumption, our implementation is asymptotically optimal.

While our techniques are not universal in the sense that one could simply "plug in" sequential object code to obtain a concurrent wait-free implementation, they do seem general enough to be easily applied to implement a variety of objects. It is worth pointing out that most universal object constructions impose a specific structure on an object, e.g., having a single object pointer that must be modified to perform an update. Our multiprocessor constructions impose no specific structure: an object can consist of any number of words linked by pointers. Because of this, we are able to design helping mechanisms that allow a process to begin helping a partially-completed operation at an intermediate point in its execution.

The rest of this paper is organized as follows. In Section 2, we present our uniprocessor implementations. Then, in Section 3, we present our multiprocessor implementations and discuss some preliminary performance results. We end with concluding remarks in Section 4. Due to space limitations, we have deferred formal proofs to the full paper. Our informal descriptions of the implementations do cover most of the formal properties needed for full proofs.

2 Uniprocessor Algorithms

In this section, we present wait-free implementations of MWCAS and linked lists for priority-based uniprocessors.

2.1 Multi-Word Compare-and-Swap

Figure 3 depicts our uniprocessor implementation of MWCAS and an associated Read primitive. The implementation requires a CAS instruction. A MWCAS operation takes as input an integer parameter indicating the number of words to be accessed, an array containing the addresses of the words to be accessed, and arrays containing old and new values for these words. We assume that process identifiers range over $\{0, \ldots, N-1\}$ and that each MWCAS operation accesses at most B words. A process reads a word by invoking the Read procedure. The words that may be accessed by MWCAS and Read are assumed to be of type *wordtype*. A word of this type consists of an application-dependent *val* field, and three fields of control information, *cnt* ($\lceil \log B \rceil$ bits), *valid* (one bit), and *pid* ($\lceil \log N \rceil$ bits).

We explain the MWCAS procedure by focusing on a MWCAS operation by process r. Such an operation is executed in three phases. In the first phase (lines 1 through 14), the kth word that is accessed by r — call it w — is updated so that its *val* field contains the desired new value, the *cnt* field contains the value k, the *valid* field is false, and the *pid* field contains the value r (see lines 11 and 12). In addition, the old value of w is saved in the shared variable $Save[r, k]$ (line 8). The *pid* and *cnt* fields of w are used by other processes to retrieve the old value from the *Save* array. The *pid* field is also used as an index into the *Status* array, the role of which is described below.

To understand the "effect" the first phase has on the words that are accessed, it is necessary to understand how each word's "current value" is defined. Let w denote a variable of type *wordtype* that is accessible by a MWCAS or Read operation. Then, the *current value* of w, denoted $Val(w)$, is defined as follows.

	val	*cnt*	*valid*	*pid*	
x:	12	2	*true*	2	$Val(x) = 12$
y:	3	1	*false*	3	$Val(y) = 22$
z:	8	3	*true*	4	$Val(z) = 8$

Save[3, 1]: 22 *Status*[3]: 0

(a)

	val	*cnt*	*valid*	*pid*	
x:	5	0	*false*	4	$Val(x) = 12$
y:	10	1	*false*	4	$Val(y) = 22$
z:	17	2	*false*	4	$Val(z) = 8$

Status[3]: 0 *Save*[3, 1]: 22 *Status*[4]: 0

Save[4, 0]: 12 *Save*[4, 1]: 22 *Save*[4, 2]: 8

(b)

	val	*cnt*	*valid*	*pid*	
x:	5	0	*true*	4	$Val(x) = 5$
y:	10	0	*true*	4	$Val(y) = 10$
z:	17	0	*true*	4	$Val(z) = 17$

Status[3]: 1 *Save*[3, 1]: 22 *Status*[4]: 2

(c)

	val	*cnt*	*valid*	*pid*	
x:	12	2	*true*	2	$Val(x) = 12$
y:	3	1	*false*	3	$Val(y) = 22$
z:	56	4	*true*	9	$Val(z) = 56$

Status[3]: 0 *Save*[3, 1]: 22 *Status*[4]: 1

(d)

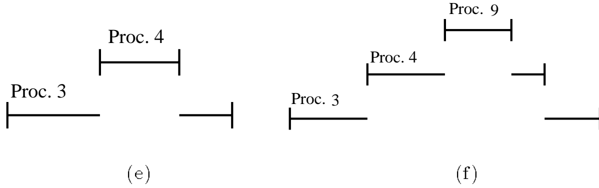

Figure 4: Process 4 performs a `MWCAS` operation on words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. The contents of relevant shared variables are shown (a) at the beginning of the operation; (b) after the loop in lines 3..14; (c) at the end of the operation, assuming success; and (d) at the end of the operation, assuming failure on word z. The interleavings that result in (c) and (d) are shown in (e) and (f), respectively.

$$Val(w) = \begin{cases} w.val & \text{if } w.valid \ \vee \ Status[w.pid] = 2 \\ Save[w.pid,\ w.cnt] & \text{otherwise} \end{cases}$$

Observe that $Val(w)$ depends on the value of $Status[r]$ if $w.pid = r$. $Status[r]$ is initialized to 0 when a `MWCAS` operation of r begins (line 1). If the operation is interfered with by other `MWCAS` operations, or if the current value of some word accessed by the operation differs from the old value specified for that word, then $Status[r]$ is assigned the value 1 (lines 10, 13, 19, and 21). If $Status[r] = 2$, then process r's latest `MWCAS` operation has succeeded.

With the definition of $Val(w)$ in mind, the "effect" of the first phase of a `MWCAS` operation can now be understood. This phase does not change the current value of any word that is accessed. However, if this phase is "successful" — i.e., $Status[r]$ is not assigned the value 1 by any process — then at the end of the first phase, the proposed new value for each word is contained within the *val* field of that word. The second phase of a `MWCAS` operation consists of only one statement: the `CAS` at line 15. This `CAS` attempts to both validate and commit the operation by resetting the value of $Status[r]$ from 0 to 2. This `CAS`, if successful, atomically changes the current value of each accessed word to the desired new value. The third and final phase consists of lines 16 through 22. In this phase, each word w that is accessed by the `MWCAS` operation of r is "cleaned up" so that $w.pid \neq r \ \vee \ w.valid$ holds. This implies that the current value of word w does not depend on $Status[r]$. Hence, when process r performs a subsequent `MWCAS` operation, reinitializing $Status[r]$ does not change the current value of any word.

Lines 19 and 21 are executed to invalidate any pending lower-priority `MWCAS` operation that has been interfered with. Note that such a pending operation exists for word w if process r detects that $w.valid$ is false. Line 19 is executed by process r only if its own operation has succeeded in changing the value of some word. Line 21 is executed by process r only if its attempt to "clean up" a word fails. This failure signifies that that word has been modified by a higher-priority process during r's execution, so it is appropriate to invalidate any pending lower-priority operation that accesses that word.

Example. Figure 4 depicts the effects of a `MWCAS` operation m by process 4 on three words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. Inset (a) shows the contents of various variables just before m begins. Note that the current value of each word matches the desired old value. Inset (b) shows relevant variables after the first phase of m has completed, assuming no interferences by higher-priority processes. The current value of each word is unchanged. Note that changing the value of $Status[4]$ from 0 to 2 in inset (b) would have the effect of atomically changing the current value of each of x, y, and z to the desired new value. Inset (c) shows relevant variables at the termination of m, assuming no interferences by higher-priority processes. The current value of each word is now the desired new value, and all *valid* fields are *true* (so the value of $Status[4]$ is no longer relevant). Before returning, process 4 updates $Status[3]$ (line 19 of Figure 3) to indicate that process 3 (which must be of lower priority) has been interfered with. Inset (d) shows relevant variables at the termination of m, assuming an interference on word z by process 9 (which must be of higher-priority) with new value 56. $Status[4]$ is now 1, indicating the failure of process 4's operation. $Status[3]$ is left unchanged in this case. Observe that process 4 has successfully restored the original values of words x and y. Insets (e) and (f) show the operation interleavings corresponding to insets (c) and (d), respectively. □

Our uniprocessor `MWCAS` implementation is disjoint access parallel [9] and is *much* simpler and more efficient than previous disjoint access parallel algorithms for asynchronous systems [2, 5, 9]. One disadvantage of our implementation is that certain bits within each accessed word must be reserved for control information (the *cnt*, *valid*, and *pid* fields). The

```
type  nodeptr = record  ptr: pointer to nodetype;  bit: 0..1  end;
      nodetype = record  key: keytype;  val: valtype;  next: nodeptr  end;
      anntype = record  ptr: pointer to nodetype;  pid: 0..N  end
      partype = record  node: pointer to nodetype;  key: keytype;  op: (ins, del, sch)  end

shared variable
      First, Last: nodetype;                                   /* List's sentinel nodes */
      Par: array [0..N − 1] of partype;       /* Par[p] stores parameters to process p's operation */
      Ann: anntype    /* Variable in which current operation is announced; Ann.pid = N when no operation is pending */
      Rv: array [0..N] of 0..2            /* Return values: 0 means pending; 1 means false; 2 means true */

initially First = (−∞, 0, (&Last, 0)) ∧ Last = (∞, 0, (NIL, 0))

private variable                                        /* For process p, where 0 ≤ p < N */
      pid: 0..N;  curr: pointer to nodetype;  nextp, nextnextp: nodeptr;  key, nextkey: keytype

procedure Insert(key: keytype; val: valtype)
1:   Par[p].node := nodealloc();
2:   *Par[p].node := (key, val, (NIL, 0));
3:   Par[p].key := key;
4:   Par[p].op := ins;
5:   Do_op()

procedure Delete(key: keytype)
6:   Par[p].key := key;
7:   Par[p].op := del;
8:   Par[p].node := NIL;
9:   Do_op();
10:  nodefree(Par[p].node)

procedure Search(key: keytype)
11:  Par[p].key := key;
12:  Par[p].op := sch;
13:  Do_op();
14:  return(Rv[p] = 2)

procedure Do_op()
15:  pid := Ann.pid;
16:  if pid < N ∧ Rv[pid] = 0 then
17:        Help(pid)
     fi;
18:  Rv[p] := 0;
19:  Ann.ptr := &First;
20:  Ann.pid := p;
21:  Help(p);
22:  Ann.ptr := &First;
23:  Ann.pid := N

procedure Findpos(key: keytype; pid: 0..N) returns pointer to nodetype
24:  while Rv[pid] = 0 do
25:        curr := Ann.ptr;
26:        nextp := curr−>next;
27:        nextkey := nextp.ptr−>key;
28:        if Rv[pid] ≠ 0 ∨ nextkey ≥ key ∨ nextp.ptr = &Last then
29:              return(curr)
           fi;
30:        Ann.ptr := nextp.ptr
     od;
31:  return(&First)

procedure Help(pid: 0..N − 1)
32:  key := Par[pid].key;
33:  curr := Findpos(key, pid);
34:  nextp := curr−>next;
35:  nextkey := nextp.ptr−>key;
36:  nextnextp := nextp.ptr−>next;
37:  if Rv[pid] = 0 then
38:     case Par[pid].op of
39:     ins: new := Par[pid].node;
40:          if nextkey ≠ key then
41:             CAS(&(new−>next), (NIL, 0), (nextp.ptr, 0));
42:             CAS(&(curr−>next), nextp, (nextp.ptr, 1));
43:             nextp.bit := 1;
44:             if Rv[pid] = 0 then
45:                 CAS(&(curr−>next), nextp, (new, 0))
46:             else CAS(&(curr−>next), nextp, (nextp.ptr, 0))
             fi  fi
47:     del: if nextkey = key then
48:             CAS(&(curr−>next), nextp, (nextnextp.ptr, 0));
49:             Par[pid].node := nextp.ptr
          fi
50:     sch: if nextkey ≠ key then Rv[pid] := 1; return fi
        esac;
51:     Rv[pid] := 2
     fi
```

Figure 5: Wait-free implementation of linked-lists for priority-based uniprocessors.

multiprocessor implementation of Section 3.1 does not require such control information and could be applied within a uniprocessor system. However, this implementation requires CCAS and is not disjoint access parallel.

2.2 Linked Lists

We now turn our attention to the linked-list implementation for priority-based uniprocessors shown in Figure 5. We begin our description of this implementation by explaining the shared variables that are used. *First* and *Last* are sentinel nodes at the beginning and end of the list, respectively. $Par[p]$ is used to record the parameters of a list operation by process p. $Par[p].node$ stores the address of a node that is to be (has been) inserted (deleted). $Par[p].key$ is used to store the key of a node to be deleted or searched for. $Par[p].op$ records the type of operation in progress. The implementation is based on the idea of incremental helping, as discussed in the introduction and depicted in Figure 2. The shared variable *Ann.pid* is used to record the process currently being helped. *Ann.pid* equals N when there is no process to help. In our list implementation, care is taken to ensure that work is not repeated by multiple processes when helping. All list operations require a scan of the list, and the shared variable *Ann.ptr* records the last node successfully scanned. By using *Ann.ptr*, we can avoid restarting a scan at the beginning of the list when helping a partially completed operation. $Rv[p]$ records the return value of process p's latest list operation.

The implementation consists of six procedures, *Insert*, *Delete*, *Search*, *Do_op*, *Help*, and *Findpos*. The first three of these procedures are invoked to perform a list operation. Each performs some operation-specific initiatization and then calls *Do_op*. In *Do_op*, a check is made to see if there exists a (lower-priority) process with an announced operation that is unfinished (line 16). If such an operation exists, it is helped by calling *Help*. Then, the process calling *Do_op* announces its own operation (lines 18-20) and calls *Help* to execute it. Before returning, *Ann* is updated to indicate that no process needs help (lines 22 and 23).

As mentioned above, each list operation requires a scan of the list. This scan is performed by *Findpos*. A process p invokes $Findpos(k, r)$ to help perform the scan associated with an operation of process r. The scan attempts to locate the predecessor of the first node in the list whose key is at least k. During an iteration of the loop at lines 24-30, the last node successfully scanned during the current operation is read (line 25). A pointer to this node is returned if the key of its successor is at least k, or if the scan reaches the end of the list (line 28-29). Otherwise, the successor is recorded as the last node successfully scanned (line 30). If $Rv[r]$ becomes nonzero at any point during the scan, then r's operation

must have been completed by some higher-priority process. In this case, *Findpos* returns a pointer to *First*, ensuring that the pointers can be safely dereferenced in lines 34-36.

Most of the work that is required to execute an operation is performed by the *Help* procedure. Consider a process p that invokes *Help* to help some process r (p and r could be the same process). Suppose that p reads key k in line 32 and that its subsequent call to *Findpos* returns a pointer to node m, whose successor n is the first node with key at least k. These steps and some additional initialization are performed in lines 32-36. After all initialization has been performed, a check is made to determine if r's operation has been completed by some higher-priority process (line 37). If the operation has not been completed, then p attempts to complete it by executing one of the three cases in lines 38-50. The case of search is simple: $Rv[r]$ is simply updated to indicate whether node n's key is k (line 50). We now consider insert and delete.

In the case of an insert operation, the new node to be inserted is first read (line 39). If the key is not already in the list, then the *next* field of the new node is made to point to n (line 41). Then, the *bit* field of m's *next* field is changed from 0 to 1 without modifying the pointer (line 42). If r's operation is not completed by a higher-priority process before line 44, then p attempts to complete r's insert by swapping in the new node and resetting m's *bit* field (line 45). Note that if a high-priority process preempts p and helps r after line 42 but before line 45 or 46, it will reset m's *bit* field before relinquishing the processor, ensuring that p's `CAS` at either line 45 or 46 fails. To complete the explanation, we must show that if p is preempted by a process that helps r between lines 37 and 42, then the `CAS` operations at lines 41 and 42 do not corrupt the list. If p is preempted prior to line 41, then by the time it resumes execution, the node it is attempting to insert is either in the linked list or the free list (it may have been deleted by some higher-priority process). In either case, this node's *next* pointer is not *NIL* (assuming the free list is implemented with sentinels), so the `CAS` at line 41 has no effect. If p is preempted prior to line 42, then when it resumes execution, the `CAS` at line 42 will attempt to set the *bit* field of the node p is trying to insert. However, p will subsequently fail the test at line 44 and clear this *bit* field at line 46 (if it has not been cleared already by a higher-priority process).

In the case of a delete operation, if the key of node n is k, then the *next* field of node m is assigned the value of the *next* field of n (line 48). In addition, the address of the deleted node is stored in $Par[r].node$ (line 49) so that process r may subsequently deallocate this node (line 10). To complete the explanation of delete, we must show that if p is preempted between lines 37 and 48, then the `CAS` at line 48 does not corrupt the list. Note that this `CAS` can succeed only if n is m's successor. Now, suppose that process p is preempted before executing its `CAS` at line 48, and suppose that while it is preempted, node n is deleted and subsequently reinserted as m's successor. It would seem that p could potentially delete n when it resumes execution, corrupting the list. Fortunately, this cannot happen, because a node cannot be reinserted until it has been deallocated by the process that deletes it (line 10) and subsequently reallocated by the process wanting to insert it (line 1). In the case under consideration, node n will not be deallocated until process r executes line 10, which is after p returns from the *Help* procedure.

Like our uniprocessor `MWCAS` implementation, our uniprocessor list implementation requires control information to be packed within a word (in this case, the *bit* field of *nodeptr*). Our multiprocessor list implementation does not require such control information and could be applied within a uniprocessor system (at the expense of having to use `CCAS`).

3 Multiprocessor Algorithms

In this section, we present wait-free implementations of `MWCAS` and linked lists for priority-based multiprocessors. As mentioned previously, our multiprocessor implementations use `CCAS`. In Section 3.3, we show that if `CAS` is available, then `CCAS` can be easily implemented in just a few instructions. Figure 8(a) in Section 3.3 defines the semantics of `CCAS`.

3.1 Multi-Word Compare-and-Swap

Our `MWCAS` implementation for multiprocessors is shown in Figure 6. We begin our description of this implementation by considering the shared variables that are used. $Ann[R]$ is the announce variable for processor R. Incremental helping is used on each processor, so only one announce variable per processor is required. $Ann[R]$ equals N when there is no process to help on processor R. $Par[p]$ is used to store parameters associated with `MWCAS` operations of process p. These parameters include the number of words that are accessed, addresses of these words, and lists of old and new values. $Rv[p]$ is set to 0 when a `MWCAS` operation of process p begins, is set to 1 when the compare phase of such an operation is completed, and is set to 2 or 3 when such an operation completes. A value of 2 (3) signifies a return value of true (false). The shared variable V is a compare-only version number that is passed to `CCAS`. It consists of a counter field *cnt* and a boolean field *needhelp*. $V.cnt$ is assumed to not cycle during any wait-free operation. The implementation is based on the idea of cyclic helping described in the introduction. With this helping scheme, processors are considered in turn, as if they formed a logical ring. A "help counter" is used to indicate the current processor under consideration. The value of the help counter is given by $V.cnt \ mod \ P$. P here is defined to be the total number of processors in the system. When the help counter is advanced to point to processor R, $V.needhelp$ is set to true iff there is a process on processor R that needs to be helped. A process is allowed to help a process on processor R only if it detects that $(V.cnt \ mod \ P = R) \wedge V.needhelp$ holds. Thus, the decision whether or not to help a process on processor R is fixed when the help counter is advanced to point to R. Since this decision is made atomically when the help counter is advanced, there can be no disagreement among processes as to whether a process on processor R should be helped.

Process p performs a `MWCAS` operation by calling the `MWCAS` procedure. After some initialization (lines 1-2), two rounds of cyclic helping are performed (lines 4-14). During the first round, p repeatedly advances the help counter until any pending announced (lower-priority) operation on its processor has been completed. It then announces its own operation (line 14) and performs a second round of cyclic helping in order to complete its own operation. The loop at lines 6-13 performs one round of cyclic helping. The test at line 8 causes the loop to terminate once the currently-announced operation on p's processor has been completed and the help counter has been advanced. If the help counter points to a processor that has a process that needs help, then the *Help* procedure is invoked at line 9. Lines 10-13 advance the help counter to the next processor on the logical ring. Line 15

```
type  wdlist = array[0..B − 1] of wdtype;                                  /* List of old and new values for MWCAS */
      addrlist = array[0..B − 1] of pointer to wdtype;                    /* Addresses to perform MWCAS on */
      partype = record numwds: 1..B; addr: addrlist; old: wdlist; new: wdlist end;
      vertype = record cnt: 0..C − 1; needhelp: boolean end                /* These fields are stored in one word */

shared variable
      Ann: array [0..P − 1] of 0..N;  /* Ann[R] is announce variable for processor R; equals N if no currently announced operation on R */
      Par: array [0..N − 1] of partype;                                    /* Par[p] stores parameters to operations of process p */
      Rv: array [0..N] of 0..3;    /* Rv[p] stores process p's status and return value: 0 means compare phase not completed; ... */
                    /* ... 1 means compare phase completed but swap phase not completed; 2 means MWCAS returns true; ... */
                                                                /* ... 3 means MWCAS returns false; Rv[N] always equals 2 */
      V: vertype                                  /* V.cnt is the version number; V.cnt mod P is the help counter; ... */
                                   /* ... V.needhelp indicates if help is needed on processor currently pointed to */

initially (∀i : 0 ≤ i < P :: Ann[i] = N) ∧ Rv[N] = 2

private variable                                   /* For process p, where 0 ≤ p < N running on processor mypr */
      pid, cpid, nxthelp: 0..N;  /* pid is a process on the same processor; cpid is current process to help; nxthelp is next process to help */
      ver: vertype; par: pointer to partype; i : 0..1

procedure MWCAS(numwds: 1..B; addr: addrlist; old, new: wdlist)
1:   Par[p] := (numwds, addr, old, new);
2:   Rv[p] := 0;
3:   for i := 0 to 1 do
4:       pid := Ann[mypr];
5:       if pid < N then
6:           while true do
7:               ver := V;
8:               if Rv[pid] ≥ 2 ∧ (ver.cnt mod P ≠ mypr ∨
                                          ¬ver.needhelp) then break fi;
9:               if ver.needhelp then Help(ver) fi;
10:              nxthelp := Ann[ver.cnt + 1 mod P];
11:              if nxthelp = N ∨ Rv[nxthelp] ≥ 2 then
12:                  CAS(&V, ver, ((ver.cnt + 1) mod C, false));
13:              else CAS(&V, ver, ((ver.cnt + 1) mod C, true));
                 fi
             od
         fi;
14:      Ann[mypr] := p
     od;
15:  Ann[mypr] := N

procedure Help(ver: vertype)
16:  cpid := Ann[ver.cnt mod P];
17:  if Rv[cpid] ≥ 2 then return fi;
18:  par := &Par[cpid];
19:  for i := 0 to par−>numwds − 1 do
20:      if *par−>addr[i] ≠ par−>old[i] then
21:          if ¬CCAS(&V, ver, &Rv[cpid], 0, 3) then break fi ;
22:          return
         fi
     od;
23:  CCAS(&V, ver, &Rv[cpid], 0, 1);
24:  for i := 0 to par−>numwds − 1 do
25:      if V ≠ ver then return fi;
26:      if Rv[cpid] ≥ 2 then return fi;
27:      if par−>old[i] ≠ par−>new[i] then
28:          CCAS(&V, ver, par−>addr[i], par−>old[i], par−>new[i])
         fi
     od;
29:  CCAS(&V, ver, &Rv[cpid], 1, 2);
30:  return
```

Figure 6: Wait-free implementation of `MWCAS` for priority-based multiprocessors.

sets the announce variable on p's processor to indicate that no process currently requires helping.

The *Help* procedure is called to help a `MWCAS` operation m of some process q that is executing on the processor that is pointed to by the help counter. It can be shown that the *Help* procedure is invoked at most P times during each round of cyclic helping ($2P$ times in total). Lines 19-22 are executed to check to see if the current value of each word that is accessed by m matches the desired old value. If a mismatch is detected, then $Rv[q]$ is set to 3 (line 21); otherwise, $Rv[q]$ is set to 1 (line 23). Lines 24-29 are executed only if no mismatch was detected. For each word this is accessed by m, line 28 is executed to assign the desired new value to that word. As an optimization, line 28 is executed only if the specified old and new values differ. The `CCAS` operation of line 29 is executed if m successfully completes.

It should be obvious from the description above that each `MWCAS` operation takes time proportional to $2 \cdot P \cdot T$, where T is the time required for the longest operation. This bound applies to all operations, irrespective of priority. In some applications, it may be advantageous to treat operations of higher-priority processes with greater urgency. For such applications a variation of cyclic helping called *priority helping* can be used. With priority helping, the help counter is always advanced to the processor with the highest-priority pending operation. Implementing this scheme requires a few straightforward changes to lines 10-13. In addition, $Ann[R]$ needs to be modified to hold both the identity of the current process to help on processor R and the priority of the currently-running process on processor R. (Note these may be different processes. If a process p on processor R helps a lower-priority process q on R, then p must first record its own priority in $Ann[R]$. Otherwise, if there are other announced operations on other processors with priority greater than q's but less than p's, p may be delayed unnecessarily. This is very similar to priority inheritance in real-time systems [11].) Advancing the help counter in this scheme requires an $O(P)$ scan of the announce array. It can be easily shown that, with priority helping, if an operation is of highest priority, then at most two other concurrent operations can be completed before it (one on its own processor and one on another processor).

`MWCAS` primitives are usually used in conjunction with read operations, and we have yet to mention how such operations could be implemented. Using our implementation, if a `MWCAS` operation is performed on a set of words, then those words are updated in sequence. Suppose a process p reads two variables X and Y in sequence, and these reads overlap the execution of some `MWCAS` operation m of another process q that updates both X and Y. It might be possible for the read of X to obtain the new value written by m, while the read of Y obtains Y's old value. In this case, m does not "appear" atomic to process p.

There are three solutions to this problem. The first is to do nothing. `MWCAS` is usually used in the following way: first, a collection of words is read; then, some local computation is performed; finally, the words that were read are updated using `MWCAS`. If an old value is read (incorrectly) from some word, then the subsequent `MWCAS` operation simply fails. No real harm is done, although performance could be impacted (in most lock-free and wait-free algorithms these steps would be repeated until the `MWCAS` succeeds). The second solution to the scenario above is to try to force the read of X to return X's old value. This requires making old values available while a `MWCAS` operation is in progress, which complicates the implementation. In addition, given the way `MWCAS` is usually

```
type  nodetype = record  key: keytype;  val: valtype;  next: pointer to nodetype  end;
      anntype = record  ptr: pointer to nodetype;  pid: 0..N  end;
      partype = record  node: pointer to nodetype;  key: keytype;  op: (ins, del, sch)  end;
      vertype = record  cnt: 0..C − 1;  needhelp: boolean  end                  /* These fields are stored in one word */

shared variable
      First, Last : nodetype;                                                                  /* List's Sentinel Nodes */
      Par: array[0..N − 1] of partype;                              /* Par[p] stores parameters to process p's operation */
      Rv: array[0..N] of 0..2;                                                /* Return values; Rv[N] always equals 2 */
      Ann: array[0..P − 1] of anntype;    /* Ann[R] is announce variable for processor R; = N if no currently announced operation on R */
      V: vertype                                  /* V.cnt is the version number; V.cnt mod P is the help counter; . . . */
                                                 /* . . . V.needhelp indicates if help is needed on processor currently pointed to */

initially First = (−∞, 0, &Last) ∧ Last = (∞, 0, NIL) ∧ (∀i : 0 ≤ i < P :: Ann[i] = (&First, N)) ∧ Rv[N] = 2

private variable                                              /* For process p, where 0 ≤ p < N running on processor mypr */
      pid, nexthelp: 0..N;  ver: vertype;  new, curr, nextp, nextnextp: pointer to nodetype;  key, nextkey: keytype;  i : 0..1

procedure Do_op()
15:  Rv[p] := 0;
16:  for i := 0 to 1 do
17:      pid := Ann[mypr].pid;
18:      if pid < N then
19:          while true do
20:              ver := V;
21:              if Rv[pid] ≠ 0 ∧ (ver.cnt mod P ≠ mypr ∨
                         ¬ver.needhelp) then break fi;
22:              if ver.needhelp then Help(ver) fi;
23:              nexthelp := Ann[(ver.cnt + 1) mod P];
24:              if nexthelp = N ∨ Rv[nexthelp] ≠ 0 then
25:                  CAS(&V, ver, ((ver.cnt + 1) mod C, false))
26:              else CAS(&V, ver, ((ver.cnt + 1) mod C, true))
                 fi
             od
         fi;
27:      Ann[mypr].ptr := &First;
28:      Ann[mypr].pid := p
     od;
29:  Ann[mypr].pid := N

procedure Findpos(key: keytype; ver: vertype; help: 0..N)
                  returns pointer to nodetype
30:  while Rv[help] = 0 do
31:      curr := Ann[ver.cnt mod P].ptr;
32:      nextp := curr−>next;
33:      if V ≠ ver then break fi;
34:      nextkey := nextp−>key;
35:      if Rv[help] ≠ 0 ∨ nextkey ≥ key ∨
                nextp = &Last then return curr fi;
36:      CCAS(&V, ver, &Ann[ver.cnt mod P].ptr, curr, nextp)
     od;
37:  return &First

procedure Help(ver: vertype)
38:  pid := Ann[ver.cnt mod P].pid;
39:  key := Par[pid].key;
40:  curr := Findpos(key, ver, pid);
41:  if V ≠ ver then return fi;
42:  nextp := curr−>next;
43:  if V ≠ ver then return fi;
44:  nextnextp := nextp−>next;
45:  nextkey := nextp−>key;
46:  if Rv[pid] = 0 then
47:      case Par[pid].op of
48:      ins: if nextkey ≠ key then
49:               new := Par[pid].node;
50:               CCAS(&V, ver, &(new−>next), NIL, nextp);
51:               CCAS(&V, ver, &(curr−>next), nextp, new)
             fi
52:      del: if nextkey = key then
53:               CCAS(&V, ver, &Par[pid].node, NIL, nextp);
54:               CCAS(&V, ver, &(curr−>next), nextp, nextnextp)
             fi
55:      sch: if nextkey ≠ key then
56:               CCAS(&V, ver, &Rv[pid], 0, 1);
57:               return
             fi
         esac;
58:      CCAS(&V, ver, &Rv[pid], 0, 2)
     fi
```

Figure 7: Wait-free implementation of linked-lists for priority-based multiprocessors. *Insert*, *Delete*, and *Search* are as given in Figure 5, with "(NIL,0)" replaced by "NIL" in line 2 of *Insert*.

used, forcing a read by a process to return an old value is rather pointless, because it simply forces a subsequent MWCAS by that process to fail. The third solution is to include reads in the helping scheme. Before a read can be performed, it is only necessary that the help counter be advanced by one. This ensures that any partially-completed MWCAS is finished by the time the next read is performed. With this scheme, each read requires time proportional to $2 \cdot T$.

The implementation described in this subsection is obviously not disjoint access parallel [9], but is *much* simpler and more efficient than implementations that are [2, 5, 9]. Some limited degree of disjoint access parallelism could be achieved with our implementation by using a separate version counter for each set of MWCAS operations that may potentially transitively conflict.

3.2 Linked Lists

Our linked-list implementation for multiprocessors is shown in Figure 7. It uses many of the same mechanisms used in our previous implementations. The *Insert*, *Delete*, and *Search* are the same as defined previously in Figure 5, and the *Do_op* procedure is almost identical to the *Do_op* procedure of Figure 6.

The *Findpos* procedure of Figure 7 differs from the one in Figure 5 in only three ways. First, *curr* is updated using the announce entry of the processor currently being helped (line 31), whereas previously there was only one announce entry for all the processes. Second, a check of V has been inserted at line 33. This check is needed to ensure that *nextp* can be safely dereferenced at line 34. A process can assign *nextp* := NIL at line 32 only if the scanned node has been deallocated and subsequently set to NIL in line 2 of the *Insert* procedure by another process. However, the scanned node can be deallocated only after V has been incremented. Third, the appropriate announce entry is updated using CCAS (line 36) rather than by means of a simple write. This ensures that a "late" update by a process that was preempted and then resumed has no effect.

The changes to the *Help* procedure in Figure 5 are similar to those described for *Findpos*: checks of V have been inserted (lines 41 and 43 of Figure 7) and updates are now performed using CCAS. Since CCAS is now being used, there is no need for the *bit* field that is updated in lines 41-46 of

```
procedure CCAS(···)
⟨ if *V ≠ ver then return false fi;
  if *X ≠ old then return false fi;
  *X := new;
  return true ⟩
             (a)
```

```
procedure CCAS(···)
 1: x := *X;
 2: if x ≠ old then return false fi;
 3: ⟨⟨ if *V ≠ ver then return false fi;
 4:    return(CAS(X, x, (new.val, x.cnt + 1)))⟩⟩
                    (b)
```

```
procedure CCAS(···)
 1: if *X ≠ old then return false fi;
 2: ⟨⟨ if *V ≠ ver then return false fi;
 3:    return(CAS(X, old, new))⟩⟩
                    (c)
```

Figure 8: Parameters to CCAS are CCAS(V: **pointer to integer**; *ver* : **integer**; X: **pointer to** *valtype*; *old*, *new* : *valtype*), where *valtype* is some single-word type. (a) Definition of CCAS. (b) Implementation using small counter field. (c) Implementation using delays.

Figure 5. As a result, the code for the insert case actually becomes a bit simpler.

3.3 Implementing Conditional Compare-and-Swap

In this subsection, we show how to implement the CCAS instruction. This instruction is defined in Figure 8(a). The angle brackets in this figure indicate that CCAS is atomic. As the figure shows, CCAS is a restriction of the more well-known two-word compare-and-swap (CAS2) instruction in which one word is a compare-only version number (given by V in the figure). This version number is incremented after each wait-free operation and is assumed to not cycle during any single operation, i.e., it can be viewed as an unbounded integer.[5] Unfortunately, as mentioned in the introduction, CAS2 is not widely available in hardware and is difficult to implement in software. However, CCAS appears to be much easier to implement than CAS2. Figures 8(b) and 8(c) give two possible implementations.

In Figure 8(b), the $*X$ parameter is tagged with a small counter field. X is assumed here to point to a shared variable that is updated within the cyclic or priority helping schemes by means of CCAS operations — it is *only* updated by such operations. The double angle brackets around lines 3 and 4 indicate that these lines are executed without preemption. This could be ensured in practice by either disabling interrupts or by having the operating system roll back a process to line 3 if it is preempted at line 4. The counter $X{-}{>}cnt$ is assumed to be large enough so that it does not cycle during a single round of helping (i.e., helping one operation) or between the execution of lines 3 and 4 by any process. Note that, because lines 3 and 4 are executed without preemption, only a small number of bits for $X{-}{>}cnt$ are required (e.g., on an 8-processor machine, three or four bits would probably suffice). This is obviously important, since the counter is being packed within a word.

It is clearly in accordance with the semantics of CCAS to return from either line 2 or line 3 or to return *false* from line 4, so consider the case in which the CAS at line 4 returns *true*. Note that line 4 is reached only after passing the test at line 3. In our object implementations, passing this test signifies that other processes have not yet advanced the help counter to another processor. Because $X{-}{>}cnt$ cannot cycle during a single round of helping or between lines 3 and 4, it follows that the CAS at line 4 succeeds only when it should.

All of our object implementations have the property that a variable $*X$ that is modified by means of CCAS operations is assigned a sequence of distinct values during a single round of helping. Thus, it is really not necessary to define $X{-}{>}cnt$ to be large enough so that it does not cycle in one round. For applications with this property, it is possible to go a step further and completely eliminate the *cnt* field. This is accomplished by inserting a $delay(\Delta)$ statement after any code that attempts to increment $*V$, where Δ in the worst-case execution time of lines 3 and 4 in Figure 8(b). With this change, if $*V = ver$ holds when line 3 is executed, then $*X$ cannot be modified between lines 3 and 4 by any process engaged in a round of helping where $*V > ver$. (In our object implementations, the $delay(\Delta)$ statement is not even necessary: enough code is executed between any increment of $*V$ and subsequent CCAS that modifies $*X$ to ensure that at least Δ time units have passed.) The result is the CCAS implementation of Figure 8(c). A very desirable property of this implementation is that it does not require certain bits of $*X$ to be reserved for control information.

[5] In our implementations, the version number and a control bit (*needhelp*) are packed together in one word. However, for ease of explanation, we simply consider $*V$ to be an integer here.

3.4 Performance Results

We have conducted preliminary performance experiments that compare our multiprocessor wait-free linked-list implementation to a lock-free list implementation presented recently by Greenwald and Cheriton [7]. We chose their implementation to test against because it is among the most efficient linked-list algorithms currently known. The experiments we conducted were performed on a four-processor SGI-R10000 machine. The priority-based preemption model was simulated at the user level by inserting predefined preemption points into each process. At a preemption point, a process randomly decides whether to relinquish its processor. The code was designed to ensure that preemptions occur in a priority-based manner. It is important to note that simulating priority-based preemptions in this way only approximates the behavior of a true priority-based system.

Our algorithm uses CCAS and Greenwald and Cheriton's uses CAS2, while the SGI-R10000 provides only CAS. However, we were able to efficiently implement CCAS and CAS2 using the approach given in Figure 8(c). (In the lock-free list implementation, CAS2 is used in manner that is similar to how CCAS is used.) In the version of our algorithm that was tested, the *Findpos* procedure was optimized to perform a CCAS instruction once for every 100 nodes scanned.

In our experiments, the total time for the processes to perform a total of 50,000 insertion/deletion operations on a sorted list was measured. Runs were obtained for various list sizes ranging from 200 to 2,000 elements. (For smaller lists, we were not able to get accurate timing figures because computation times were too short compared to the granularity of the timing routines.) The total time required for our algorithm was typically 1.5 to 2 times higher than that required for the lock-free algorithm (with 1.5 being more typical). Another lock-free list implementation, which uses only CAS, was recently proposed by Valois [13]. Although we did not test against Valois' algorithm, Greenwald and Cheriton report that their algorithm is faster than his algorithm by a factor of about ten under high contention [7]. Based on this, we believe that our algorithm would perform better than Valois' algorithm on a priority-based system.

The fact that our algorithm loses to Greenwald and Cheriton's algorithm does not come as a surprise, because in their algorithm, each operation executes as a very simple lock-free

retry loop, which can be expected to require little overhead in the average case. It is important to keep in mind, however, that our algorithm is wait-free, whereas Greenwald and Cheriton's algorithm is only lock-free. This has important implications for real-time systems (one of the primary targets of our work). In such systems, tasks must be guaranteed to meet their deadlines, and such guarantees require that tight worst-case execution times for object accesses be known. With lock-free algorithms, determining such bounds is not easy, because accurately bounding the cost of retry-loop interferences that can occur *across* processors is difficult.

To get some sense of worst-case performance, we modified the lock-free algorithm to keep track of the worst-case number of retries of any retry loop in a run. Worst-case values of 10 to 30 retries were common. Worst-case values of 30 to 50 retries were less common but still somewhat frequent. In contrast, with our algorithm implemented on four processors, the time for each operation is at most eight times that of an interference-free operation. Given the synthetic nature of our experiments and the way in which we simulated preemptive scheduling, the retry figures given here should be viewed with healthy skepticism. Nonetheless, we believe that it is reasonable to conclude the following from our results: (i) the worst-case performance of our algorithm is very good; (ii) the average-case performance of our algorithm is reasonable (although not as good as the fastest-known algorithm).

Recent results from a related paper [1] indicate that in real-time systems, our algorithms may exhibit performance that is better than is possible when applying them in a non-real-time system as done here. In particular, it is shown in [1] that if our cyclic helping scheme is used in conjunction with a real-time scheduler, then each operation requires only one traversal of the helping ring, instead of two. In addition, simulations reported in [1] indicate that in real-time multiprocessor systems, priority helping is very effective and results in much better performance than cyclic helping. In contrast, in non-real-time systems, priority helping could result in the starvation of low-priority processes if high-priority processes perform operations very frequently.

4 Concluding Remarks

The fact that our helping schemes could be applied to implement different objects with virtually the same code is an indication that they can be generally applied in priority-based systems. Other "linear" data structures, like queues, stacks, and hash tables, are just as straightforward to implement as linked lists. More complex data structures like balanced trees require a more complicated *Help* procedure, but many of the same techniques still apply.

A major goal of our work has been to develop implementations that could be practically applied, so we have avoided algorithmic techniques that can be restrictive or inefficient when actually implemented. Our avoidance of costly techniques for ensuring disjoint access parallelism is a good example of this. As another example, we have tried to avoid packing control information into the words of a data structure wherever possible. Many published wait-free algorithms are difficult to implement because required control fields do not fit into a single memory word. We have also sought to simplify our implementations by exploiting certain characteristics of priority-based systems. Our assumption that version numbers do not cycle is a good example of this. This is reasonable for the applications targeted by our work, and having to cope with a counter that has cycled would have unnecessarily complicated our code. Another example can be found in the `CCAS` implementations given Section 3.3, where we exploit the fact that in many priority-based systems, mechanisms exist for disabling process preemptions.

Acknowledgement: We thank Michael Greenwald for his comments on an earlier draft of this paper.

References

[1] J. Anderson, R. Jain, and S. Ramamurthy, "Wait-free Object-Sharing Schemes for Real-Time Uniprocessors and Multiprocessors", manuscript, May 1997.

[2] J. Anderson and M. Moir, "Universal Constructions for Multi-Object Operations", *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 184-193.

[3] J. Anderson and M. Moir, "Universal Constructions for Large Objects", *Proc. of the Ninth Int'l Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 972, Springer-Verlag, 1995, pp. 168-182.

[4] J. Anderson and S. Ramamurthy, "A Framework for Implementing Objects and Scheduling Tasks in Lock-Free Real-Time Systems", *Proc. of the 17th IEEE Real-Time Systems Symposium*, 1996, pp. 94-105.

[5] H. Attiya and E. Dagan, "Universal Operations: Unary versus Binary", *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, 1996, pp. 223-232.

[6] B. Bershad, "Practical Considerations for Non-Blocking Concurrent Objects", *Proc. of the 13th Int'l Conference on Distributed Computing Systems*, 1993, pp. 124-149.

[7] M. Greenwald and D. Cheriton, "The Synergy Between Non-blocking Synchronization and Operating System Structure", *Proc. of the USENIX Association Second Symposium on Operating Systems Design and Implementation*, 1996, pp. 123-136

[8] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Trans. on Programming Languages and Systems*, 15(5), 1993, pp. 745-770.

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

[10] H. Massalin and C. Pu, "A Lock-Free Multiprocessor OS Kernel", Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.

[11] R. Rajkumar, *Synchronization In Real-Time Systems - A Priority Inheritance Approach*, Kluwer Academic Pubs., 1991.

[12] S. Ramamurthy, M. Moir, and J. Anderson, "Real-Time Object Sharing with Minimal System Support", *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, 1996, pp. 233-242.

[13] J. Valois, "Lock-Free Linked Lists using Compare-and-Swap", *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 214-222.