1

IMPLEMENTING HARD REAL-TIME TRANSACTIONS ON MULTIPROCESSORS

James H. Anderson, Rohit Jain, and Srikanth Ramamurthy

Department of Computer Science,
University of North Carolina at Chapel Hill

1 INTRODUCTION

Many applications exist in which hard real-time transactions must be supported; examples include embedded control applications, defense systems, and avionics systems. With the recent advent of workstation-class multiprocessor systems, multiprocessor-based implementations of these systems are of growing importance. Unfortunately, the problem of implementing hard real-time transactions on multiprocessors has received relatively little attention. With most (if not all) previously-proposed schemes for implementing such transactions, the kind of transactions that can be supported is often limited, and worst-case performance is often poor, adversely impacting schedulability.

We present here a new scheme for implementing hard real-time transactions on multiprocessors that does not place undue restrictions on the kinds of transactions that can be supported. This scheme gives rise to worst-case performance bounds that often will be lower than those arising from alternative schemes. In our approach, transactions are implemented by invoking wait-free library routines. Concurrency control is embedded within these routines, so no special support for data management is required of the kernel or from underlying server processes. These routines reduce overhead by exploiting the way transactions are interleaved on priority-based real-time systems.

Since transactions in our implementation are executed in a wait-free manner, they are not susceptible to deadlock or priority inversion, and they can be scheduled as if they were independent. In checking schedulability, the difference between the execution cost of a wait-free transaction and that of a sequential implementation of that transaction is essentially an overhead term.

This overhead term bears a resemblance to blocking factors associated with priority inversions that arise in lock-based schemes. However, these overhead terms arise for different reasons, and overhead terms when using our approach are often much smaller than those of lock-based schemes [1].

Our transaction implementation is described in detail in subsequent sections. First, in Section 2, we review previous work on implementing hard real-time transactions on multiprocessors. We then present an overview of some of the key mechanisms used in our work in Section 3. Our implementation is described in detail in Section 4. We end with concluding remarks in Section 5.

2 PREVIOUS WORK

Previous work on implementing hard real-time transactions on multiprocessors has assumed data to be memory resident. This is due to the difficulty of supporting hard deadlines in the face of unpredictable page faults and relatively long I/O latencies for disk accesses. Previous work has also focused on the use of lock-based concurrency control schemes. This is because, with optimistic schemes, transactions may abort and restart due to conflicts, and repeated restarts can cause a transaction to miss its deadline.[1]

When using lock-based concurrency control schemes on multiprocessors, mechanisms are needed to bound the effects of priority inversion. Rajkumar et al. proposed two such mechanisms by extending the uniprocessor priority ceiling protocol (PCP) [8, 10]. The resulting protocols are called the distributed PCP (DPCP) and the multiprocessor PCP (MPCP), respectively [7, 8, 9]. Both protocols assume a model in which a task can perform one or more transactions. In the DPCP, global resources are guarded by synchronization processors. A task accesses a global resource by making an RPC-like call to a global critical section (GCS) server, which executes on its behalf on the synchronization processor. The DPCP can be used in either multiprocessors or distributed systems. In contrast, the MPCP is intended for use only in shared-memory multiprocessors. In the MPCP, a task executes a GCS on its own processor by using global semaphores, which are implemented using read-modify-write instructions.

With either the MPCP or the DPCP, tasks are susceptible to very large blocking factors, and correspondingly low processor utilizations. In addition, these

[1] On uniprocessors, the number of transaction restarts over an interval of time can be bounded (assuming memory-resident data) [2]. However, bounding the effects of repeated restarts due to conflicts *across* processors in a multiprocessor system does not seem practical.

schemes require *a priori* knowledge of all transactions' resource access requirements, which makes it difficult, if not impossible, to support dynamically-generated transactions and mode changes. Large blocking factors are partially the result of the fact that high-priority tasks on one processor can be repeatedly blocked by low-priority tasks on other processors. Blocking factors are also increased by task suspensions; a task suspends itself in the DPCP, for example, whenever it accesses a GCS server. In effect, such suspensions break a task into a sequence of subtasks, each of which may experience a priority inversion.

Researchers at the University of Illinois have proposed an "end-to-end" approach for sharing resources in multiprocessors, in which tasks are converted into a sequence of periodic subtasks [4, 11]. Each subtask can perform either local computation or a single-processor transaction. Subtasks are scheduled on a per-processor basis, using uniprocessor scheduling schemes. Resources are accessed through the use of the PCP or a similar scheme. Like the DPCP, the end-to-end approach requires a binding of resources to processors. This restricts the kind of transactions that can be supported (e.g., a transaction cannot access resources mapped to two different processors). In addition, precedence constraints of subtasks are ensured by delaying the release of each subtask by an amount equal to the sum of the worst-case responses of all subtasks that must precede it. Such an approach could lead to poor schedulability. Finally, if a PCP-like scheme is used on each processor as proposed, then supporting dynamically-generated transactions and mode changes is problematic.

Lortz and Shin have implemented a hard real-time database system called MDARTS (Multiprocessor Database Architecture for Real-Time Systems), in which both RPC transactions and concurrent shared-memory-based transactions are supported [6]. This was the first (and perhaps only) actual implementation of a hard real-time multiprocessor database server with reasonable performance. However, in MDARTS, critical sections are implemented by disabling preemptions (to deal with conflicts within a processor) and by using queue-based spin locks (to deal with conflicts across processors). As a result, only transactions of very short duration can be effectively supported.

3 OVERVIEW OF OUR APPROACH

In this section, we present an overview of our approach to implementing transactions. We assume that transactions are invoked by periodic hard-deadline tasks. Each task consists of one or more phases. Each phase is either a *com-*

putation phase (which doesn't access shared data) or a *transaction phase*. All data is assumed to be memory-resident.

Before explaining the key techniques used in our transaction implementation, we first explain the notion of a "wait-free" shared object in some detail. In a wait-free object implementation, operations must be implemented using bounded, sequential code fragments, with no blocking synchronization constructs. This is often accomplished by using a *helping scheme*. Before beginning an operation, a task first "announces" its intentions by storing information about its operation in a shared "announce variable". While attempting to perform its own operation, a task also attempts to "help" other tasks with announced operations by performing their operations in addition to its own. Care must be taken to ensure that each operation is executed *exactly* once, and that a helped task can retrieve its return values from memory. This is usually accomplished by using strong synchronization primitives like compare-and-swap (`CAS`) when updating shared data so that tasks do not adversely interfere with each other. In a sense, a helping-based wait-free scheme is a "pessimistic" notion of nonblocking user-level synchronization that is similar to a lock-based scheme, with helping taking the place of blocking. (This is illustrated quite well by Figure 1, which is considered below.)[2]

In a recent paper [3], we showed that the cost of helping can be greatly reduced on priority-based real-time uniprocessor systems by using a technique called *incremental helping*. The general idea of incremental helping is illustrated in Figure 1. Before beginning a transaction, a task must first announce its intentions by updating a shared announce variable. Before a task is allowed to do this, however, it must first help any previously-announced transaction (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 task [5]. Also, with incremental helping, each task helps at most one other task, while in helping schemes for asynchronous systems, each task helps all other tasks in the worst case.

With incremental helping, the worst-case time to perform a transaction is only $2 \cdot T$, where T is the execution time of one transaction in the absence of preemption. Of the $2 \cdot T$ worst-case execution cost, a factor of T represents the overhead associated with helping. (It is possible to reduce this overhead term using priority ceiling information as explained in [1].) This overhead term is similar to blocking factors that arise in scheduling conditions when using the

[2]The "optimistic" counterpart of a wait-free object is a *lock-free* object [5]. In a lock-free object implementation, an operation may be interfered with, in which case it must be retried. Repeated retries may be needed before an operation is successful.

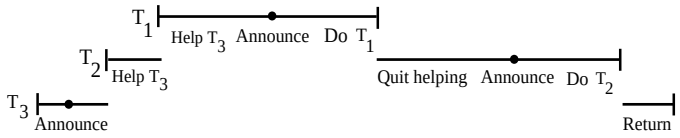

Figure 1 Task T_3 detects no previously-announced transaction, so it announces its transaction. Task T_2 preempts T_3 and begins to help T_3 to complete its transaction, but before it finishes, it is preempted by task T_1. T_1 detects that T_3's transaction is unfinished, so it too helps T_3. It then announces its own transaction, executes it, and relinquishes the processor to T_2. T_2 detects that T_3's transaction is complete, so it announces and executes its own transaction, and relinquishes the processor to T_3. T_3 detects that its transaction has been completed, so it returns.

priority inheritance protocol (PIP) [8, 10]. Like the PIP, information about which tasks access which objects is not required for implementing the helping scheme. (PCP-like schemes *do* require access information.) This is because conflicts are recorded dynamically through the use of the announce variable. The fact that object-access requirements do not have to be predeclared makes it easier to support mode changes and dynamically-generated transactions.

The notion of incremental helping can be extended for application on multiprocessors. We call the resulting scheme *cyclic helping* [1, 3]. With cyclic helping, the processors are thought of as if they were part of a logical ring. Tasks are helped through the use of a "help counter", which cycles around the ring. To advance the help counter past a processor, a task must first help the currently-announced transaction on that processor. To perform a transaction, a task first repeatedly advances the help counter until any pending (lower-priority) transaction on its own processor has been completed. It then announces its own transaction and advances the help counter until its transaction has been completed. On a ring of size P, the time to perform a transaction is therefore at most $2 \cdot P \cdot T$, where T is the largest sequential transaction cost. (Tighter worst-case bounds are given in [1].) Unlike the MPCP and DPCP, information about object-access requirements is not required for implementing cyclic helping. (However, the tighter bounds in [1] do make use of such information.)

It can often be advantageous to incorporate priority information into an object-sharing scheme. We have developed a wait-free scheme called *priority helping* that does this [1, 3]. Priority helping is similar to cyclic helping, except that the help counter is always advanced to the processor with the highest-priority pending transaction. Under priority helping, if a transaction is of highest priority, then at most one other transaction can be completed before it.

4 TRANSACTION IMPLEMENTATION

In this section, we present a detailed description of our transaction implementation. The implementation has been obtained by combining a multiprocessor-based cyclic-helping scheme with a lock-free transaction implementation for real-time uniprocessors presented previously [2]. Modifications for implementing a priority-helping scheme could be similarly applied.

Our implementation is defined by four procedures, *Exec*, *Help*, *Read*, and *Write*, which are given in Figure 3. Variable declarations that are used in these procedures are given in Figure 2. The procedures given in Figure 3 support the "illusion" of a contiguous shared array *MEM* of memory words. In reality, data is not stored in contiguous locations of memory, but is composed of a number of blocks. The *Read* (*Write*) procedure is invoked when executing a transaction to read words from (write words to) the *MEM* array. As explained below, the *Exec* procedure is called by a task to perform a transaction of that task. The *Help* procedure is invoked when one task helps another.

When a transaction of task T accesses a word in the implemented array of memory words, say $MEM[k]$, the block containing the k^{th} word is identified. If T's transaction writes into $MEM[k]$, then T must replace the corresponding block. The details of identifying blocks and replacing modified blocks are implemented within the *Read* and *Write* routines, which perform all necessary address translation and bookkeeping. These routines are called within user-supplied transaction code to read or write words of the *MEM* array; e.g., "$Write(1, Read(10))$" performs the assignment "$MEM[1] := MEM[10]$". Figure 4 shows a simple example transaction, which enqueues an item onto a shared queue. This transaction would be executed by calling $Exec(Enqueue)$.

The implemented array *MEM* is partitioned into B blocks of size S. (We assume a constant block size here for simplicity.) Figure 5 depicts this arrangement for $B = 5$. The first block contains memory locations 0 through $S - 1$, the second contains locations S through $2S - 1$, and so on. A bank of pointers — one for each block — is used to point to the blocks that make up the array. (These are really array indices, not pointers.) In order to modify the contents of *MEM*, a task makes a copy of each block to be modified, and then attempts to atomically replace the old blocks with their modified copies. The details of how this is done is explained below in conjunction with the helping scheme.

In Figure 2, *Bank* is a B-word shared array. Each element of *Bank* contains a pointer to a block of size S. The B blocks pointed to by *Bank* constitute

```
/* Assume: (at most) N tasks executing on P processors;  B, S, and C are ...
           ... as defined in the text;  version number ranges over {0, ..., D − 1} */

type
  blktype = array [0..S − 1] of memwdtype;
  wdtype = 0..B + NC − 1;
  vertype = record cnt: 0..D − 1; needhelp: boolean end /* Stored in one word */

shared variable
  Bank: array[0..B − 1] of wdtype;          /* Bank of "pointers" to array blocks */
  Blk: array[0..B + NC − 1] of blktype;     /* Array and copy blocks */
  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 */
  Ann: array[0..P − 1] of 0..N;             /* Ann[R] is the announce variable ...
       ... for processor R; equals N if no currently announced transaction on R */
  Trans: array[0..N] of pointer to function; /* Used to announce transactions */
  Stat: array[0..N] of −1..N;       /* Status of each task's latest transaction: ... */
    /* ... −1, initially; in [0, ..., N − 1] after first phase; N, after second phase */
  Addrlist: array[0..N − 1, 0..B − 1] of pointer to wdtype; /* Parameters for ... */
  Oldlist, Newlist: array[0..N − 1, 0..B − 1] of wdtype;    /* ... CCAS's */
  Numblks: array[0..N − 1] of 0..B          /* Number of CCAS's to perform */

initially (∀k : 0 ≤ k < B :: Bank[k] = NC + k ∧ Blk[NC + k] = (initial value of
                  block k)) ∧ (∀n : 0 ≤ n < P :: Ann[n] = N) ∧ Stat[N] = N

private variable                                              /* For task T_p */
  copy: array[0..C − 1] of wdtype;          /* Indices for copy blocks of T_p */
  curr: array[0..B − 1] of wdtype;          /* T_p's current view of MEM */
  blklist: array[0..B − 1] of 0..B − 1;     /* List of accessed blocks */
  dirty: array[0..B − 1] of 0..2;   /* 1 if block read, 2 if modified, 0 otherwise */
  mypr: 0..P − 1;                           /* Task T_p's processor */
  hid, lid, wid: 0..N;   /* Task identifiers: hid is task to help; lid is local task ...
                        on T_p's processor; wid is "winning" task from first phase */
  ver: vertype;                             /* A value read from V */
  addr: pointer to wdtype;                  /* Address to perform CCAS on */
  old, new: wdtype;                         /* Old and new value for performing CCAS */
  env: jmp_buf;                             /* Used by setjmp and longjmp system calls */
  dirtycnt: 0..C − 1;  m: 0..1;  i, j, numblks, blk: 0..B;  ret: boolean

initially (∀k : 0 ≤ k < C :: copy[k] = pC + k) ∧ (∀k : 0 ≤ k < B :: dirty[k] = 0)
```

Figure 2 Variable declarations.

the current version of the *MEM* array. We assume that an upper bound C is known on the number of blocks modified by any transaction. Because a task's

```
procedure Exec(tr: pointer to function)
1:  Trans[p] := tr;
2:  Stat[p] := −1;
3:  for m := 0 to 1 do
4:    lid := Ann[mypr];
5:    if lid < N then
6:      while true do
7:        ver := V;
8:        if Stat[lid] = N ∧ (ver.cnt mod P ≠ mypr ∨ ¬ver.needhelp) then break fi;
9:        if ver.needhelp then Help(ver) fi;
10:       hid := Ann[ver.cnt + 1 mod P];
11:       if hid = N ∨ Stat[hid] = N then
12:         CAS(&V, ver, ((ver.cnt + 1) mod D, false));
13:       else CAS(&V, ver, ((ver.cnt + 1) mod D, true));
          fi
        od
      fi;
14:   Ann[mypr] := p
    od;
15: Ann[mypr] := N

procedure Help(ver: vertype)
16: hid := Ann[ver.cnt mod P];
17: if Stat[hid] = N then return fi;
18: if Stat[hid] = −1 then
19:   dirtycnt, numblks := 0, 0;
20:   if setjmp(env) ≠ 1 then
21:     ∗Trans[hid]();
22:     for j := 0 to numblks − 1 do
23:       Newlist[p, j] := curr[blklist[j]]
        od;
24:     Numblks[p] := numblks;
25:     ret := CCAS(&V, ver, &Stat[hid], −1, p);
26:     i := 0;
27:     for j := 0 to numblks − 1 do
28:       if dirty[blklist[j]] = 2 ∧ ret then
29:         copy[i] := Oldlist[p, j];
30:         i := i + 1
          fi;
31:       dirty[blklist[j]] := 0
        od
    fi  fi;
    /∗ Help continued ∗/
32: wid := Stat[hid];
33: if wid = −1 ∨ wid = N then
      return
    fi;
34: numblks := Numblks[wid];
35: for j := 0 to numblks − 1 do
36:   if V ≠ ver then
        return
      fi;
37:   if Stat[hid] = N then
        return
      fi;
38:   addr := Addrlist[wid, j];
39:   old := Oldlist[wid, j];
40:   new := Newlist[wid, j];
41:   CCAS(&V, ver, addr, old, new)
    od;
42: CCAS(&V, ver, &Stat[hid], wid, N)
```

Figure 3 Transaction implementation. For each procedure, p is the index of the invoking task. All private variables refer to those of T_p.

```
procedure Read(mwd : 0..BS − 1)
        returns memwdtype
43: if V ≠ ver then
44:   longjmp(env, 1)
    fi;
45: if Stat[hid] ≥ 0 then
46:   longjmp(env, 1)
    fi;
47: blk := mwd div S;
48: if dirty[blk] = 0 then
49:   dirty[blk] := 1;
50:   curr[blk] := Bank[blk];
51:   addr := &Bank[blk];
52:   Addrlist[p, numblks] := addr;
53:   blklist[numblks] := blk;
54:   Oldlist[p, numblks] := curr[blk];
55:   numblks := numblks + 1
    fi;
56: v := Blk[curr[blk]][mwd mod S];
57: return(v)
```

```
procedure Write(mwd : 0..BS − 1;
        value : memwdtype)
58: if V ≠ ver then longjmp(env, 1) fi;
59: if Stat[hid] ≥ 0 then
60:   longjmp(env, 1)
    fi;
61: blk := mwd div S;
62: if dirty[blk] = 0 then
63:   curr[blk] := Bank[blk];
64:   addr := &Bank[blk];
65:   Addrlist[p, numblks] := addr;
66:   blklist[numblks] := blk;
67:   Oldlist[p, numblks] := curr[blk];
68:   numblks := numblks + 1
    fi;
69: if dirty[blk] ≠ 2 then
70:   dirty[blk] := 2;
71:   memcpy(Blk[copy[dirtycnt]],
        Blk[curr[blk]], sizeof(blktype));
72:   curr[blk] := copy[dirtycnt];
73:   dirtycnt := dirtycnt + 1
    fi;
74: Blk[curr[blk]][mwd mod S] := value
```

Figure 3 *(continued)*

transaction copies a block before modifying it, C "copy" blocks are required per task. Therefore, a total of $B + NC$ blocks are used, where N is the number of tasks. These blocks are stored in the array *Blk*. Initially, blocks $Blk[NC]$ to $Blk[NC + B - 1]$ are the blocks of the *MEM* array, and $Blk[pC]$ to $Blk[(p + 1)C - 1]$ are task T_p's copy blocks. However, the roles of these blocks are not fixed. If T_p's copy blocks are installed as part of the *MEM* array, then T_p reclaims the replaced blocks as copy blocks (lines 26-30). Thus, some of T_p's copy blocks become part of the current array, and vice versa.

As mentioned above, user-supplied transaction code accesses the *MEM* array in a sequential manner using the *Read* and *Write* procedures. After performing a consistency check that we will describe later (lines 43-46), the *Read* procedure computes the index of the block containing the accessed word (line 47). If the block has not yet been read by the transaction invoking *Read*, then it is marked as having been read (line 49), and is recorded in the transaction's *curr* array (line 50). This array gives the transaction's "current view" of *MEM*. The block index is also recorded in an array *blklist* (line 53), which is used later

```
constant Head = n;  Tail = n + 1
private variable newtail: 0..n − 1

procedure Enqueue(input: valtype) returns {FULL, SUCCESS}
    Write(Read(Tail), input);
    newtail := (Read(Tail) + 1) mod n;
    if newtail = Read(Head) then return(FULL) fi;
    Write(Tail, newtail);
    return(SUCCESS)
```

Figure 4 Example transaction.

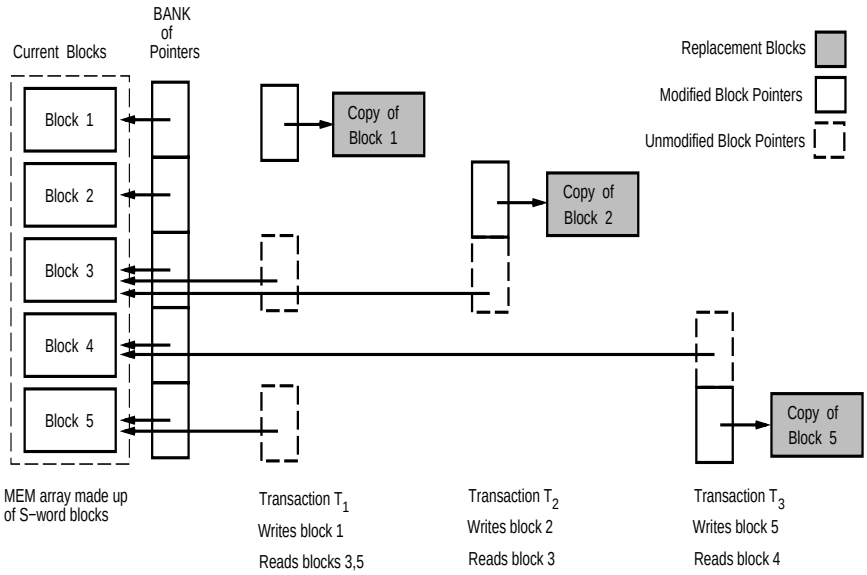

Figure 5 Implementation of the *MEM* array (depicted for $B = 5$).

in reclaiming copy blocks. In addition, the address and old value of the block pointer are saved in arrays (lines 52 and 54) that are later used to update the *MEM* array (see below). The *Read* procedure completes by retrieving a value from the appropriate offset within the block that is accessed (line 56). The *Write* procedure is similar to the *Read* procedure, except that, when a block is first modified, it is recorded as having been modified (line 70), and a local copy of the block is made (line 71).

We now explain the *Exec* and *Help* procedures. A special synchronization primitive, which we call conditional compare-and-swap (`CCAS`), is used within these procedures. `CCAS` is an atomic primitive with the following semantics.

```
procedure CCAS(V: pointer to vertype; ver: vertype;
               X: pointer to valtype; old, new: valtype) returns boolean
  if *V ≠ ver ∨ *X ≠ old then return false fi;
  *X := new;
  return true
```

As its definition 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 value. In our implementation, this word is a shared variable that stores a "version number" (the variable V — see below). `CCAS` is useful because the compare-only value can be used to ensure that a "late" `CCAS` operation by a task that has been preempted and then resumed has no effect. Fortunately, `CCAS` is easy to implement even if `CAS2` is not available. In a recent paper [3], we showed that `CCAS` can be implemented using only three high-level language statements if `CAS` (a commonly-available instruction) is available.

We continue our description of the *Exec* and *Help* procedures by considering the shared variables that are referenced in these procedures. $Ann[R]$ 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 task to help on processor R (task identifiers are assumed to range over $\{0, \ldots, N-1\}$). $Trans[p]$ is used to store a pointer to a function that implements a transaction of task T_p. $Stat[p]$ records the "status" of a transaction of task T_p. Each transaction is executed in two phases, the details of which are described below. $Stat[p]$ is initialized to -1, equals a value in $\{0, \ldots, N-1\}$ after the first phase, and equals N after the second phase.

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 transaction. Our transaction implementation is based on the idea of cyclic helping described in Section 3. 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 $V.cnt \bmod P$. 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 task on processor R that needs to be helped. A task is allowed to help a task 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 task 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 tasks as to whether a task on processor R should be helped.

As mentioned previously, *Exec* is invoked by a task T_p to perform a transaction. After some initialization (lines 1-2), two rounds of cyclic helping are performed (lines 4-14). During the first round, T_p repeatedly advances the help counter until any pending announced (lower-priority) transaction on its processor has been completed. It then announces its own transaction (line 14) and performs a second round of cyclic helping in order to complete its own transaction. 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 transaction on T_p's processor has been completed and the help counter has been advanced. If the help counter points to a processor that has a task 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 sets the announce variable on T_p's processor to indicate that no task currently requires helping.

The *Help* procedure is called to help a transaction of some task T_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). As mentioned previously, each transaction is executed in two phases. During the first phase (lines 19-31), modifications to the *MEM* array are determined by executing the user-supplied transaction code (line 21). In the second phase (lines 34-42), the *MEM* array is actually updated by performing a sequence of `CCAS` operations. When executing lines 19-24, a task T_p updates three shared arrays, $Addrlist[p]$, $Newlist[p]$, and $Oldlist[p]$, and the shared variable $Numblks[p]$. ($Addrlist[p]$ and $Oldlist[p]$ are actually updated in the *Read* and *Write* routines, which are invoked when the input transaction is executed at line 21.) The shared arrays give lists of addresses, new values, and old values to be used in performing `CCAS` operations in the second phase. Each address specifies a particular entry in *Bank*. Such an entry corresponds to a block that has been read or written while executing a transaction. The variable $Numblks[p]$ specifies the number of such blocks. Several tasks may concurrently attempt to perform the first phase of a transaction of some task T_q. One of these tasks — call it T_r — will successfully update $Stat[q]$ at line 25, signifying the end of the first phase. Task T_r reclaims blocks that have been modified as its own copy blocks for the next transaction it executes (lines 26-31). The addresses, new values, old values, and number of blocks recorded by T_r are used by all tasks that attempt to perform the second phase of T_q's transaction (lines

34-41). In this phase, each *Bank* entry corresponding to a block accessed by T_q's transaction is modified in turn. (An optimization could be added to lines 38-41 so that only blocks that are written are considered. This optimization has been omitted for brevity.) When the second phase is completed, $Stat[q]$ is updated to indicate that T_q's latest transaction is complete (line 42).

Before concluding, one subtlety that we have glossed over must be mentioned. If the *Bank* variable is modified by a task T_q during the execution of a transaction performed by some other task T_p (T_p may be executing its own transaction or that of another task), then T_p may read inconsistent values from *MEM*. Task T_p cannot possibly perform a successful `CCAS` operation in its second phase in this case, so T_p will not be able to install corrupted data. However, there is a risk that T_p's execution of the transaction code might cause an error, such as a division by zero or a range error. This problem is solved by performing a consistency check at the beginning of the *Read* and *Write* procedures (lines 43-46 and 58-60). If an inconsistency is detected, then control is returned to line 20 in the *Help* procedure using Unix-like *longjmp* calls. In this event, T_p discontinues executing the first phase of the transaction it is performing, and proceeds directly to attempt to execute its second phase (lines 32-42).

5 CONCLUDING REMARKS

Many optimizations of the basic implementation just described are possible. For example, with a slight modification, read-only transactions can be executed with greater concurrency, which is very desirable if only a small percentage of transactions are updates. This involves adding a $(P+1)^{st}$ "virtual" processor to the cyclic helping ring. While the help counter points to the virtual processor, any read-only transaction can be performed without helping other transactions. Also, the cyclic helping scheme actually can be implemented in a manner that requires a transaction to perform only one traversal of the helping ring instead of two [1]. Details of these optimizations are omitted here due to space limitations.

The explanation of our transaction implementation would seem to imply that one centralized help counter is always required. This is not the case. Transactions often can be grouped into classes such that transactions in different classes cannot possibly conflict with each other. Only one help counter per class is required in this case. Transactions in different classes may execute with complete concurrency. Moreover, the helping ring for a class really only needs to include processors with tasks that execute transactions in that class.

The results above leave many opportunities for further research. Of foremost importance are experimental studies that compare our transaction implementation with more conventional ones. Such studies require a real-time multiprocessor testbed. We are currently developing such a testbed at UNC.

REFERENCES

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

[2] J. Anderson, S. Ramamurthy, M. Moir, K. Jeffay, "Lock-Free Trans. for Real-Time Systems", *Proc. First Int'l Workshop on Real-Time Databases: Issues and Applications*, 1996, 107-114.

[3] J. Anderson, S. Ramamurthy, R. Jain "Implementing Wait-Free Objects on Priority-Based Systems", *Proc. 16th ACM Symp. on Prin. of Distr. Comp.*, to appear.

[4] R. Bettati, *End-to-End Scheduling to Meet Deadlines in Distributed Systems*, Ph.D. Thesis, Computer Science Dept., Univ. of Illinois, 1994.

[5] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Trans. on Prog. Langs. and Sys.*, 15(5), 1993, 745-770.

[6] V. Lortz, *An Object-Oriented Real-Time Database System for Multiprocessors*, Ph.D. Thesis, Computer Science Dept., Univ. of Michigan, 1994.

[7] R. Rajkumar, "Real-Time Synchronization Protocols for Shared Memory Multiprocessors", *Proc. Int'l Conf. on Distr. Comp. Sys.*, 1990, 116-123.

[8] Raghunathan Rajkumar, *Synchronization In Real-Time Systems - A Priority Inheritance Approach*, Kluwer Academic Publications, 1991.

[9] R. Rajkumar, L. Sha, J. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", *Proc. IEEE Real-Time Sys. Symp.*, 1988, 259-269.

[10] L. Sha, R. Rajkumar, J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time System Synchronization", *IEEE Trans. on Computers*, 39(9), 1990, 1175-1185.

[11] J. Sun, R. Bettati, J. W.-S. Liu, "Using End-to-End Scheduling Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems", *Proc. 11th IEEE Workshop on Real-Time Op. Sys. & Software*, 1994.