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

Abstract

We present a new approach to implementing real-time transactions on memory-resident data on shared-memory multiprocessors. This approach allows hard deadlines to be supported without undue overhead. 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 the overhead involved in executing transactions by exploiting the way transactions are interleaved on priority-based real-time systems. We present evidence that shows that our approach often entails substantially less overhead than more conventional priority inheritance schemes.

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

*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.

little attention. As explained in the next section, with most (if not all) previously proposed schemes for implementing such transactions, the kinds of transactions that can be supported is often severely limited, and worst-case performance (either actual or estimated) is often very poor, adversely impacting schedulability.

In this paper, we present 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 can be expected to be much lower than those arising from alternative schemes in many applications. 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 the overhead involved in executing transactions by exploiting the way transactions are interleaved on priority-based real-time systems. As a result, they perform much better than previous implementations of wait-free transactions proposed for asynchronous systems [1].

Since transactions in our implementation are executed in a wait-free manner, they are not susceptible to deadlock or priority inversion, and schedulability can be checked using scheduling conditions for independent tasks. In applying these conditions, 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 in many applications, overhead terms when using our approach can be expected to be much smaller than blocking factors when using lock-based schemes. The avoidance of locks in our implementations has other important consequences. For example, there is no need for kernel-level information (e.g., priority ceilings) used for dealing with priority inversions. As a result, mode changes are simplified because kernel-level information does not have to be recalculated. Also, a measure of fault-tolerance is provided: a failed transaction cannot cannot hold locks on any object and therefore cannot prevent object accesses by other transactions.

Our transaction implementation is described in detail in subsequent sections. Before presenting this description, we first review previous work on implementing hard real-time transactions on multiprocessors. This review appears in Section 2. After reviewing previous work, we present an overview of some of the key mechanisms used in our work in Section 3. Our transaction implementation is then described in detail in Section 4. In

our approach, transactions are performed using a novel synchronization primitive, which we call *conditional compare-and-swap*. We explain how to implement this primitive from simpler primitives in Section 5. We end with a brief discussion of some preliminary performance results and other concluding remarks in Section 6.

2 Previous Work

Previous work on implementing hard real-time transactions on multiprocessors has assumed data to be memory resident and has focused on the use of lock-based concurrency control schemes. Data is assumed to be memory-resident because the unpredictability of page faults and the relatively long I/O latencies for disk accesses make supporting hard deadlines impractical. In this paper, we consider only memory-resident real-time databases. Lock-based concurrency control schemes have been the focus of previous work because, with optimistic schemes, transactions may abort and restart due to conflicts, and repeated restarts can cause a transaction to miss its deadline. On uniprocessors, the number of transaction restarts over an interval of time can be bounded (assuming memory-resident data) [3], and thus hard-deadline transactions *can* be supported. However, bounding the effects of repeated restarts due to conflicts *across* processors in a multiprocessor system does not seem practical.

When using lock-based concurrency control schemes on multiprocessors, mechanisms are needed to bound the effects of priority inversion. The most well-known solution to this problem in uniprocessors is the priority ceiling protocol (PCP) [15, 16]. To ensure predictability in multiprocessor systems, Rajkumar et al. proposed the distributed PCP (DPCP) and the multiprocessor PCP (MPCP), both of which extend the PCP [12, 13, 14]. Both approaches 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. Synchronization is achieved by using read-modify-write instructions to obtain global semaphores, which reside in shared memory.

With either the MPCP or the DPCP, tasks are susceptible to very large blocking factors, and correspondingly low processor utilizations. In addition, these 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, and each subtask may experience a priority inversion. In contrast, an instance of a task can experience at most one priority inversion when using the PCP on a uniprocessor. To the best of our knowledge, neither the DPCP nor the MPCP has been implemented in real systems. This is certainly due to the fact that these schemes give rise to such large blocking overheads.

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 using heuristics [7, 17]. Each subtask can perform either local computation or a single-processor transaction. Heuristics are used to deduce appropriate release times and deadlines for the subtasks of a task based on the timing requirements of that task. 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. The end-to-end approach is based on the observation that, since subtasks are executed sequentially, there is no need to execute accesses to remote global resources at a higher priority level than local tasks. Like the DPCP, the end-to-end approach requires a binding of resources to processors. This complicates the job of assigning the components (tasks and resources) of an application to processors, and restricts the kind of transactions that can be supported (e.g., a transaction cannot access resources mapped to two different processors). In addition, the heuristics that are applied to determine subtask release times and deadlines can be overly pessimistic for certain task sets, leading to poor schedulability. Finally, if a PCP-like scheme is used to access resources on a processor in the end-to-end approach 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 [11]. This was the first (and perhaps only) actual implementation of a hard real-time multiprocessor database server with reasonable performance. However, concurrency control in MDARTS is somewhat limiting. 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

```
type Qtype = record data: valtype; next: pointer to Qtype end
shared variable Head, Tail: pointer to Qtype
private variable old, new: pointer to Qtype; addr: pointer to pointer to Qtype

procedure Enqueue(input: valtype)
    *new := (input, NULL);
    repeat old := Tail;
           if old ≠ NULL then addr := &(old->next) else addr := &Head fi
    until CAS2(&Tail, addr, old, NULL, new, new)
```

Figure 1: Lock-free enqueue implementation.

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. In the remainder of the paper, we assume that transactions are invoked by periodic tasks. For simplicity, we assume that all tasks are subject to hard deadlines. Each task consists of one or more phases. Each phase is either a *computation 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 combining a "retry loop" structure with a "helping" scheme. If a retry loop structure is used *without* a helping scheme, then the resulting object implementation is called "lock-free". Figure 1 depicts a lock-free enqueue operation that is implemented in this way. An item is enqueued in this implementation by using a two-word compare-and-swap (`CAS2`) instruction[1] to atomically update a tail pointer and either the "next" pointer of the last item in the queue or a head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the `CAS2` instruction succeeds. An important property of lock-free implementations such as this is that operations may repeatedly *interfere* with each other. An interference results in the queue example when a successful `CAS2` by one task results in failed `CAS2` by another task.

As mentioned above, wait-free objects are often implemented by adding a helping scheme to a retry loop

[1] The first two parameters of `CAS2` specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed.

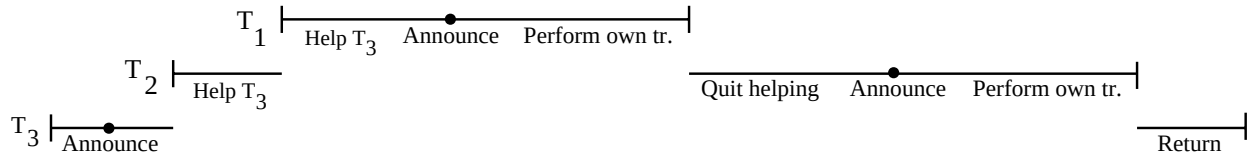

Figure 2: Task T_3 detects no previously announced task, so it announces its transaction. Before T_3 can complete its transaction, it is preempted by task T_2. Task T_2 begins to help T_3 to complete its transaction, but before it finishes, it is preempted by task T_1. Task T_1 detects that T_3's transaction has been announced but is not finished, so it too helps T_3. It then announces its own transaction, executes it, and relinquishes the processor to task T_2. Task T_2 detects that T_3's transaction is complete, so it announces its own transaction, executes it, and relinquishes the processor to T_3. Task T_3 detects that its transaction has been completed, so it returns. Note that the overall execution is similar to what one might find in a lock-based system, with helping taking the place of blocking.

structure. Before beginning an operation, a task first "announces" its intentions by storing information about its operation in a shared "announce variable". While in its retry loop, a task attempts to "help" other tasks with announced operations by performing their operations in addition to its own. If a task experiences repeated interferences, then its operation is eventually completed by another task. Care must be taken to ensure that each operation is executed *at most* once, and that a helped task can retrieve its return values from memory. With most helping implementations that have been proposed [1, 2, 9], performance is at least linear in the number of tasks sharing an object.

In a recent paper [4], 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 2. 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 [1, 2, 9]. In addition, 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. Thus, this scheme scales well with transaction size and the number of tasks in the system (in fact, its performance is *independent* of the latter). Of the $2 \cdot T$ worst-case execution cost, a factor of T represents the overhead associated with helping. This overhead term is similar

to blocking factors that arise in scheduling conditions when using priority inheritance schemes. However, with incremental helping, kernel overhead is avoided. Like the priority inheritance protocol, information about which tasks access which objects is not required, either for implementing the helping scheme or for applying schedulability tests to tasks. (PCP-like schemes *do* require such information.) This is because conflicts are recorded dynamically through the use of the announce variable, and because worst-case time bounds do not depend on access information. Because object access requirements do not have to be predeclared, incremental helping can be used to handle hard-deadline transactions that are generated dynamically.

The notion incremental helping can be extended for application on multiprocessors. The resulting scheme, which we call *cyclic helping*, is illustrated in Figure 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 from processor R to the next processor on the ring, a task must first help the currently announced task on processor R. In order to perform a transaction, a task does the following: it first repeatedly advances the help counter until any pending announced (lower-priority) transaction on its own processor has been completed; it then announces its own transaction; it then repeatedly advances the help counter until is own transaction has been completed. With cyclic helping implemented on a P-processor system, the time to perform a transaction is proportional to $2 \cdot P \cdot T$, where T is the time required for the longest transaction. Thus, this scheme also scales well with transaction size and the number of tasks in the system. For task sets with even a modest number of objects shared across processors, $2 \cdot P \cdot T$ would typically be *much less* than corresponding blocking factors that arise when using the MPCP or the DPCP. In addition, the MPCP and the DPCP require object access requirements to be predeclared, while cyclic helping does not.

The cyclic helping scheme just described is very similar to the circulation of a token in a token ring system, where to perform a transaction, a task first requests the token. One may wonder whether the same technique could be applied in a lock-based system. When implementing objects on a real-time multiprocessor, the main problem to be faced is that of ensuring that worst-case object access times are reasonably short in the face of untimely task preemptions. In a lock-based implementation, the progress of the token could be stalled if it is "held" by a preempted task. If the preempted task has only partially executed its critical section, then it must resume execution before the token can be forwarded. This requires a mechanism by which a task requesting the token on one processor may force a preempted task on another processor to resume execution. We know of

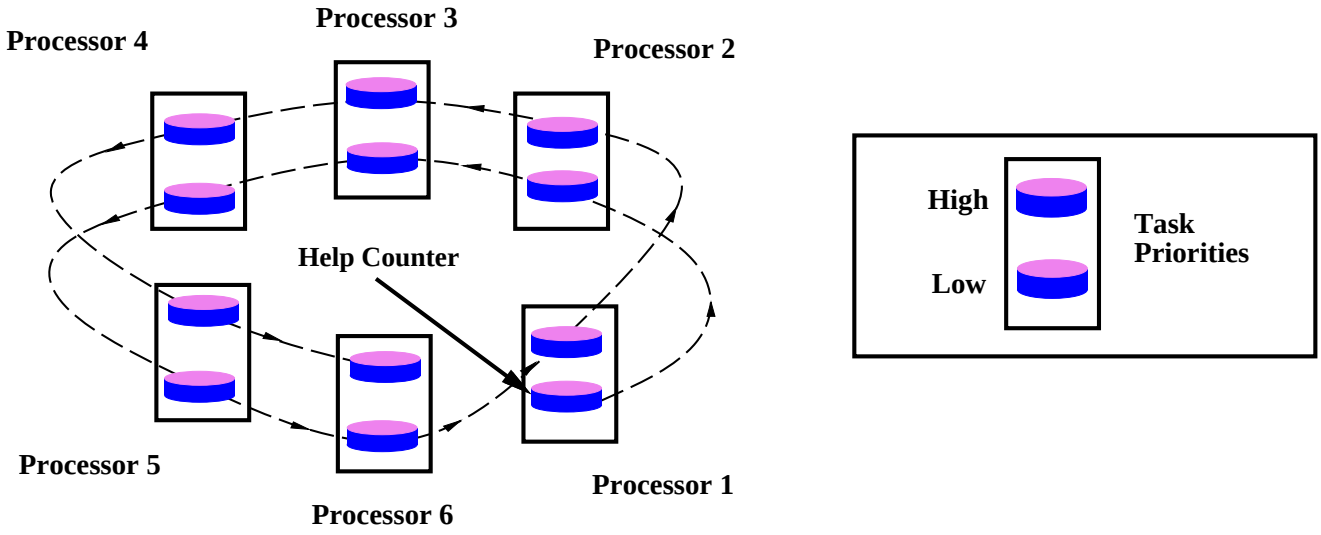

Figure 3: Cyclic helping. Processors are arranged in a logical ring. A "help counter" indicates the next processor on which a transaction will be helped. To perform a transaction, the ring must be traversed twice in the worst case (as depicted).

no way to efficiently implement such a mechanism. With our cyclic helping scheme, the requesting task itself ensures that the "token" makes progress around the ring.

As the discussion above shows, there is an interesting connection between wait-free and lock-based object-sharing schemes from a schedulability standpoint. In a sense, wait-free synchronization is a "pessimistic" notion of nonblocking user-level synchronization, and lock-free synchronization is its "optimistic" counterpart.

4 Transaction Implementation

In this section, we present a detailed description of our transaction implementation. The implementation is based on a lock-free transaction implementation for real-time uniprocessors presented previously in [3]. This previous implementation has been modified for application on shared-memory multiprocessors by using a cyclic helping scheme.

Our implementation is defined by four procedures, *Exec*, *Help*, *Read*, and *Write*, which are given in Figure 5. Variable declarations that are used in these procedures are given in Figure 4. The procedures given in Figure 5 support the "illusion" of a contiguous shared array *MEM* of memory words. In reality, data is not stored in

```
/* Assume: (At most) N tasks executing on P processors;  MEM array consists of B blocks of S words each; ...
                                                  ... each task has C copy blocks;  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                    /* These fields are 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 */
    Status: array[0..N] of −1..N;                 / Status[p] is used to announce the status of task T_p's latest transaction: ... */
        /* ... = −1 if first phase is not complete; ∈ [0, ..., N − 1] if first phase is complete; = N if second phase is complete */
    Addrlist: array[0..N,0..B − 1] of pointer to wdtype;                        /* List addresses for CCAS's in second phase */
    Oldlist, Newlist: array[0..N − 1,0..B − 1] of wdtype;                       /* List of old and new values for CCAS's in second phase */
    Numblks: 0..B                                                               /* Number of words to perform CCAS on in second phase */

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

private variable                                                                /* For task T_p */
    copy: array[0..C − 1] of wdtype;                                            /* Indices for copy blocks of task T_p */
    curr: array[0..B − 1] of wdtype;                                            /* Task T_p's current view of the MEM array */
    blklist: array[0..B − 1] of 0..B − 1;                                       /* List of blocks that have been accessed */
    dirty: array[0..B − 1] of 0..2;                                             /* 0 if block not accessed, 1 if read, 2 if modified */
    mypr: 0..P − 1;                                                             /* Task T_p's processor */
    hid, lid, wid: 0..N;          /* Task identifiers: hid is the id of a task to help; lid is the id of a local task on T_p's processor; ...
                                                  ... wid is the id of the task that "wins" first phase by successfully updating Status */
    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 */
    dirtycnt: 0..C − 1; m: 0..1; i, j, numblks, blk: 0..B; ret: boolean;         /* Counters, loop indices, block indices, return value */
    env: jmp_buf                                                                /* Used by setjmp and longjmp system calls */

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

Figure 4: Variable declarations.

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. A pointer to a function is passed as an input parameter. This function is assumed to contain sequential code implementing the transaction to be executed. 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 $MEM[x]$, the block containing the x^{th} word is identified. If T's transaction writes into $MEM[x]$, then T must replace the corresponding block. The details of identifying blocks and replacing modified blocks are hidden from the programmer by means of the *Read* and *Write* routines, which perform all necessary address translation and bookkeeping. These routines are called within the programmer's transaction code in order to read or write a

```
procedure Exec(tr: pointer to function)
1:  Trans[p] := tr;
2:  Status[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 Status[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 ∨ Status[hid] = N 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: hid := Ann[ver.cnt mod P];
17: if Status[hid] = N then return fi;
18: if Status[hid] = −1 then
        /* Perform first phase */
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, &Status[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;
32: wid := Status[hid];
33: if wid = −1 ∨ wid = N then return fi;
    /* Perform second phase */
34: numblks := Numblks[wid];
35: for j := 0 to numblks − 1 do
36:     if V ≠ ver then return fi;
37:     if Status[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, &Status[hid], wid, N)
    fi
```

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

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

Figure 5: Transaction implementation. For each procedure, p is the index of the invoking task. All private variables refer to those of T_p.

```
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 6: Example transaction.

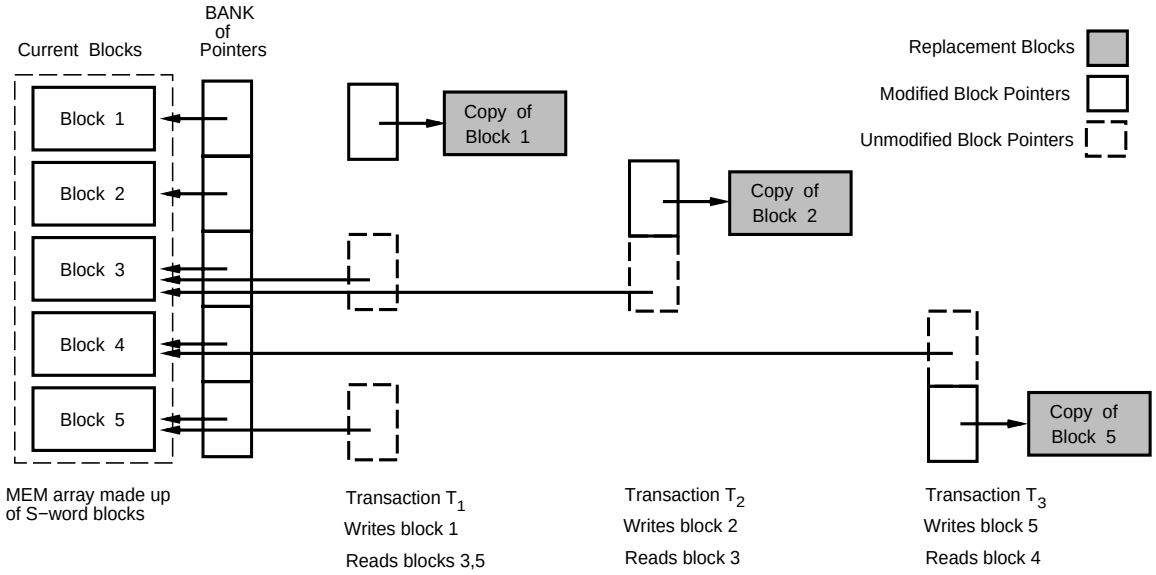

Figure 7: Implementation of the *MEM* array for lock-free transactions (depicted for $B = 5$).

word of the *MEM* array. Thus, instead of writing "$MEM[1] := MEM[10]$", the programmer would write "$Write(1, Read(10))$". Figure 6 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 7 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 4, *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 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 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 43-44), the *Read* procedure computes the index of the block containing the accessed word (line 45). If the block has not yet been read by the transaction invoking *Read*, then it is marked as having been read (line 47), and is recorded in the transaction's *curr* array (line 48). This array gives the transaction's "current view" of *MEM*. The block index is also recorded in an array *blklist* (line 51), which is used later in reclaiming copy blocks. In addition, the address and old value of the block pointer are saved in arrays (lines 50 and 52) 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 54). 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 67), and a local copy of the block is made (line 68).

We now explain the *Exec* and *Help* procedures. Synchronization is achieved in these procedures by using two primitives: compare-and-swap (`CAS`) and conditional compare-and-swap (`CCAS`). `CAS` is widely available, but `CCAS` is not. `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; it's semantics is formally defined in Figure 8(a). The version number is incremented after each transaction and is assumed to not cycle during any single transaction.[2] This ensures that a "late" `CCAS` operation executed by a preempted task has no effect. Unfortunately, `CAS2` is directly provided on only a few existing processors (e.g., Motorola 68030 and 68040 processors). Algorithms for

[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.

implementing `CAS2` from simpler primitives *are* known [1, 5, 10], but none are efficient enough to be practically applied. An efficient hardware-based implementation of `CAS2` was recently proposed by Greenwald and Cheriton [8], 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 [8] for details). Fortunately, as shown in Section 5, `CCAS` appears to be much easier to implement than `CAS2`. If `CAS` is available, then `CCAS` can be implemented in just a few instructions.

We continue our description of the *Exec* and *Help* procedures by considering the shared variables that are referenced in these procedure. $Ann[R]$ 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. $Status[p]$ records the "status" of a transaction of task T_p. Each transaction is executed in two phases, the details of which are described below. $Status[p]$ is initialized to -1, is modified to hold a value in $\{0, \ldots, N-1\}$ when the first phase completes, and is assigned the value N when the second phase completes.

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 $V.cnt \ mod \ P$. 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 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. It should be noted that, due to the helping scheme employed, 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 *Status*[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, *Status*[q] is updated to indicate that T_q's latest transaction is complete (line 42).

Before concluding this description, 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 executed 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 the *MEM* array. 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 43-44 and 56-57). If an inconsistency is detected, then control is returned from the *Read* or *Write* procedure 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).

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. If updates are rare, then the help counter would almost always point to the virtual processor, and read-only transactions would execute with very little synchronization overhead. This scheme penalizes updates, but only slightly (the cyclic ring has grown, but only by one processor). Details of this and other optimizations are omitted here due to space limitations.

5 Implementing Conditional Compare-and-Swap

We now explain how to implement `CCAS`. The semantics of `CCAS` is formally 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 transaction and is assumed to not cycle during any single transaction, i.e., it can be viewed as an unbounded integer.[3] Unfortunately, as mentioned in the previous section, `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

[3] In our transaction implementation, 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.

```
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.

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 during a round of cyclic or priority helping 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 task to line 3 if it is preempted at line 4. The counter $X\text{->}cnt$ is assumed to be large enough so that it does not cycle during a single round of helping or between the execution of lines 3 and 4 by any task. Note that, because lines 3 and 4 are executed without preemption, only a small number of bits for $X\text{->}cnt$ are required (e.g., on an 8-processor machine, three or four bits would probably suffice). This is obviously important, since the counter value 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 transaction implementation, passing this test signifies that other tasks have not yet advanced the help counter to another processor. Because $X\text{->}cnt$ 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.

Our transaction implementation has 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 $X\text{->}cnt$ 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 task engaged in a round of helping where $*V > ver$. (In our transaction implementation, 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.

6 Concluding Remarks

The explanation of our transaction implementation would seem to imply that one centralized help counter is always required when implementing a real-time database system. This is not the case. In many applications, transactions 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 help counter for a class really only needs to cycle around the set of processors with tasks that execute transactions in that class; this set may include fewer than the total number of processors.

The research outlined above leaves many opportunities for further research. Of foremost importance are experimental studies that compare our wait-free transaction implementation with more conventional lock-based implementations. Such studies require a real-time multiprocessor testbed. We are currently developing such a testbed at UNC, but it is not yet operational. Simulation studies are an alternative to experimentation. However, simulation studies will be definitive only if they are conducted using meaningful parameter values. The parameters that affect schedulability when using wait-free and lock-based transaction implementations are very different. Wait-free schemes are sensitive to such factors as copying overhead and the cost of performing synchronization instructions. Lock-based schemes are sensitive to such factors as context switching costs and kernel overhead. It is very difficult to select meaningful values for such parameters without examining actual implementations, which again requires an operational testbed.

Despite our current limitations in conducting definitive performance studies, we have conducted a preliminary study to evaluate the effectiveness of cyclic helping. In this study, a wait-free linked-list implementation based on cyclic helping presented by us elsewhere [4] was compared with a lock-free list implementation presented recently by Greenwald and Cheriton [8]. These experiments were performed on a five-processor SGI-R10000 machine. The priority-based preemption model was simulated at the user level. The wait-free list implementation that

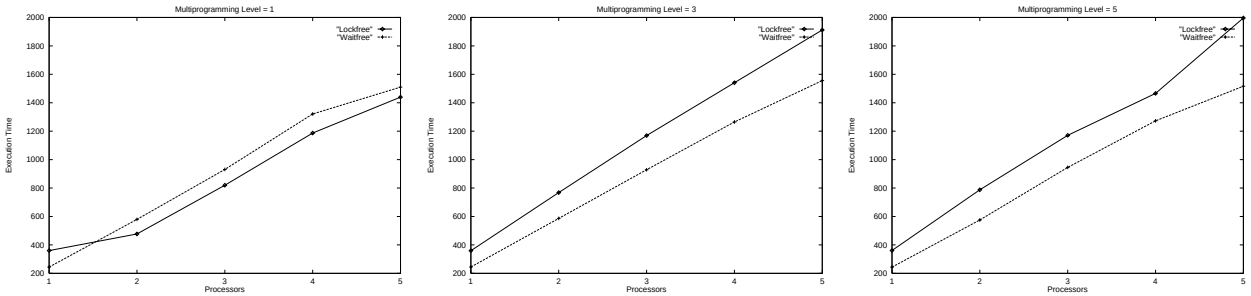

Figure 9: Comparison of wait-free and lock-free list implementations.

was tested is similar to the transaction implementation presented in this paper, with optimizations that allow list operations to be performed with no copying overhead. (Such optimizations appear to be possible for many single-object transactions.)

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 Section 5. (In the lock-free list implementation, `CAS2` is used in manner that is similar to how `CCAS` is used.)

In our experiments, the total time for the tasks to perform a total of 50,000 insertion/deletion operations on a sorted list was measured. Figure 9 depicts the relative performance (in tens of milliseconds) of the two implementations for different multiprogramming levels. These figures indicate that, as the multiprogramming level is increased, our algorithm outperforms Greenwald and Cheriton's. Our algorithm also outperforms theirs as the length of the list is increased, although graphs showing this have been omitted due to space limitations. Another lock-free list implementation, which uses only `CAS`, was recently proposed by Valois [18]. 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 three under low contention, and about ten under high contention [8]. Greenwald and Cheriton also reported that their algorithm outperformed a fast spin lock algorithm with backoff. We believe it is highly doubtful that OS-based multiprocessor locking protocols like the MPCP and the DPCP could exhibit performance close to that of either our algorithm or Greenwald and Cheriton's.

While our algorithm does not *dramatically* outperform Greenwald and Cheriton's, one should keep in mind that our algorithm is wait-free, whereas their algorithm and Valois' algorithm are only lock-free. This has important implications for hard real-time systems. 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 interferences that can occur *across* processors is difficult. With wait-free algorithms, worst-case execution times are easily computed. The only other means of implementing wait-free linked lists that we know of is to use universal constructions [2, 9]. Although we did not test against such constructions, they entail very high helping overhead. As a result, they would likely perform much worse than our list implementation.

References

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

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

[3] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay, "Lock-Free Transactions for Real-Time Systems", *Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications*, March 1996, pp. 107-114. Expanded version to appear in *Real-Time Database Systems: Issues and Applications*, Kluwer Academic Publishers.

[4] J. Anderson, S. Ramamurthy, and R. Jain "Implementing Wait-Free Objects on Priority-Based Systems", manuscript, January 1997.

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

[6] B. Bershad, "Practical Considerations for Non-Blocking Concurrent Objects", *Proceedings of the 13th international Conference on Distributed Computing Systems*, May 1993, pp. pages 264-274.

[7] R. Bettati, *End-to-End Scheduling to Meet Deadlines in Distributed Systems*, Ph.D. Thesis, Computer Science Department, University of Illinois at Urbana-Champaign, March 1994.

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

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

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

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

[12] R. Rajkumar, "Real-Time Synchronization Protocols for Shared Memory Multiprocessors", *Proceedings of the International Conference on Distributed Computing Systems*, May 1990, pp. 116-123.

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

[14] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", *Proceedings of the IEEE Real-Time Systems Symposium*, December 1988, pp. 259-269.

[15] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time System Synchronization", *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990, pp. 1175-1185.

[16] L. Sha, R. Rajkumar, S. Son, and C. Chang, "A Real-Time Locking Protocol", *IEEE Transactions on Computers*, Vol. 40, No. 7, 1991, pp. 793-800.

[17] J. Sun, R. Bettati, and J. W.-S. Liu, "Using End-to-End Scheduling Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems", *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

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