1

LOCK-FREE TRANSACTIONS FOR REAL-TIME SYSTEMS

James H. Anderson,
Srikanth Ramamurthy,
Mark Moir and Kevin Jeffay

University of North Carolina,
Chapel Hill, North Carolina, USA

1 INTRODUCTION

Lock-free objects are an alternative to lock-based object sharing protocols such as the priority ceiling protocol [17, 21] in preemptive real-time uniprocessor systems. An object implementation is *lock-free* iff it guarantees the following: if several tasks concurrently perform operations on the object, and if some proper subset of these tasks stop taking steps, then at least one of the remaining tasks must complete its operation in a finite number of its own steps. This definition precludes the use of critical sections, because if a task stops taking steps while within a critical section, then other tasks are prevented from accessing that critical section. In several related papers, we have presented general techniques that can be used to implement lock-free objects in real-time uniprocessor systems [3, 18] and to schedule tasks that share such objects [3, 4]. Related research includes work on techniques for implementing specific lock-free objects (such as read/write buffers) [13, 19, 20], and work on synchronization mechanisms that are similar to lock-free objects but are implemented using kernel support [12, 19, 20].

Operations on lock-free objects are usually implemented using "retry loops". 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 within a retry loop to atomically update a shared tail pointer and the "next" pointer of the last item in the queue. (`CAS2` takes six parameters: the first two specify addresses of two shared variables, the next two are values to which these variables are compared, and the last two are new values to be assigned to the variables if both comparisons succeed.) The retry loop is attempted repeatedly until the `CAS2` instruction succeeds. Note that

```
type Qtype = record data : valtype; next : *Qtype end
shared variable Tail : *Qtype
private variable old, new : *Qtype
procedure Enqueue(input : valtype)
    *new := (input, NULL);
     repeat old := Tail
     until CAS2(&Tail, &(old->next), old, NULL, new, new)
```

Figure 1 Lock-free enqueue operation.

`CAS2` is used to atomically validate *and* commit an operation. An important property of lock-free implementations is that operations may *interfere* with each other. In the enqueue example, a task τ can be interfered with only if a higher-priority task performs a successful `CAS2` between τ's read of *Tail* and τ's subsequent `CAS2`.

From a real-time perspective, lock-free object implementations are of interest because they avoid priority inversion and deadlock with no underlying system support. On the surface, however, it is not immediately apparent that lock-free shared objects can be employed if tasks must adhere to strict timing constraints. In particular, repeated interferences can cause a given operation to take an arbitrarily long time to complete. Fortunately, such interferences can be bounded by scheduling tasks appropriately [4]. As explained in the next section, the key to scheduling such tasks is to allow enough spare processor time to accommodate the failed object updates due to interferences that can occur over any interval. The number of interferences within an interval is bounded by the number of task preemptions within that interval.

In this chapter, we show that previous work on lock-free objects can be extended to apply to lock-free *transactions* on memory-resident databases. Compared to conventional optimistic concurrency control schemes, our lock-free transaction implementation is most similar to optimistic concurrency control with broadcast commit [10]. The main difference between our implementation and conventional schemes is that we use a strong synchronization primitive at the user level to validate and commit transactions. The strong primitive used in our implementation is a multi-word compare-and-swap (`MWCAS`). This primitive is used as the basis for a lock-free retry loop in which operations on many objects are validated at once.

Our implementation accomplishes most of the same goals as conventional optimistic concurrency control protocols. However, our implementation requires less interprocess communication overhead than conventional client/server implementations. In addition, lock-free implementations do not require complicated recovery procedures when transactions are aborted. As with any optimistic scheme, the main overhead associated with lock-free transactions is the cost of wasted computation due to restarts.

The rest of this chapter is organized as follows. In Section 2, we review previous work on using lock-free objects in real-time systems. Then, in Section 3, we present our approach for implementing lock-free transactions. Concluding remarks appear in Section 4.

2 LOCK-FREE OBJECTS

We begin this section by reviewing previous work on scheduling hard real-time tasks that share lock-free objects. We then consider the issue of hardware support for lock-free synchronization.

2.1 Scheduling with Lock-Free Objects

Although lock-free objects do not give rise to priority inversions, it may seem that unbounded retry loops render such objects useless in real-time systems. Nonetheless, Anderson et al. have shown that if tasks on a uniprocessor are scheduled appropriately, then such loops are indeed bounded [4]. We now explain why such bounds exist.

For the sake of explanation, let us call an iteration of a retry loop a *successful update* if it successfully updates an object, and a *failed update* otherwise. Thus, an invocation of a lock-free operation consists of any number of failed updates followed by one successful update. Consider two tasks τ_i and τ_j that access a common lock-free object B. Suppose that τ_i causes τ_j to experience a failed update of B. On a uniprocessor, this can happen only if τ_i preempts the access of τ_j and then updates B successfully. Thus, there is a correlation between failed updates and task preemptions. The maximum number of task preemptions within a time interval can be determined from the timing requirements of the tasks. This gives a bound on the number of failed updates in that interval. A

task set is schedulable if there is enough free processor time to accommodate the failed updates that can occur over any interval.

In [4], scheduling conditions are established for the DM [14] and EDF [15] priority assignments. In order to state these conditions, we must first define some notation. Each condition applies to a collection of N periodic tasks $\{\tau_1, \ldots, \tau_N\}$. The period of task τ_i is denoted by p_i, and the relative deadline of task τ_i is denoted by l_i, where $l_i \leq p_i$; under the EDF scheme, we assume $l_i = p_i$. Tasks are labeled in nondecreasing order by deadline, i.e., $l_i < l_j \Rightarrow i < j$. Let c_i denote the worst-case computational cost (execution time) of task τ_i when it is the only task executing on the processor. Let s denote the execution time for one loop iteration in the implementation of a lock-free object. For simplicity, all such loops are assumed to have the same cost. Note that s is also the extra computation required in the event of a failed update. Given this notation, sufficient scheduling conditions for the DM and EDF schemes can be stated.

Theorem 1: (Sufficiency under DM) *A set of periodic tasks that share lock-free objects on a uniprocessor can be scheduled under the DM scheme if, for each task τ_i, there exists some $t \in (0, l_i]$ such that*

$$(\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s) \ \leq \ t.$$

Informally, this condition states that a task set is schedulable if, for each job of every task τ_i, there exists a point in time t between the release of that job and its deadline, such that the demand placed on the processor in the interval between the job's release and time t is at most the available processor time in that interval. Demand in this interval can be broken into two components: demand due to job releases, ignoring failed object updates (this is given by the first summation); and demand due to failed object updates, which is bounded by the number of preemptions by higher-priority tasks in the interval (this is given by the second summation). In comparing the above condition to the DM condition for independent tasks given in [5], we see that the above condition essentially requires that the computation time of each task be "dilated" by the time it takes for one lock-free loop iteration.

Theorem 2: (Sufficiency under EDF) *A set of periodic tasks that share lock-free objects on a uniprocessor can be scheduled under the EDF scheme if*

$$\sum_{j=1}^{N} \frac{c_j + s}{p_j} \leq 1.$$

This condition states that a task set is schedulable if processor utilization is at most 1. As in the case of DM scheduling, this condition extends the corresponding condition for independent tasks [15] by requiring that the computation time of each task be dilated by the cost of one lock-free loop iteration.

The results presented above suggest a general strategy for determining the schedulability of tasks that share lock-free objects. First, determine a bound on demand due to failed updates over any interval of time. Then, modify scheduling conditions for independent tasks by incorporating this demand. Scheduling conditions derived in this manner are applicable not only for tasks that perform single-object updates, but also for tasks that perform multi-object transactions.

The bounds on failed updates given in the theorems above are based on the assumption that the cost of each lock-free retry loop equals that of the largest such loop, which is reasonable if retry loop costs are fairly uniform. For cases in which large variations in loop costs exist, Anderson and Ramamurthy have shown that linear programming can be applied to obtain much tighter scheduling conditions [3]. In this approach, the total cost of failed updates in τ_i and higher-priority tasks over an interval $\mathcal{I}$ is first expressed as a linear expression involving a set of variables; each variable represents the number of interferences of a particular retry loop as caused by a particular task in $\mathcal{I}$. Then, a set of conditions constraining the variables is derived. A simple example of such a constraint is that the total number of interferences caused by task τ_j in $\mathcal{I}$ is bounded by the number of job releases of τ_j in $\mathcal{I}$. Finally, an upper bound on the total cost of interferences in τ_i and higher-priority tasks during $\mathcal{I}$ is calculated using linear programming. This approach can be used to derive scheduling conditions for most common scheduling schemes.

2.2 Hardware Support

A possible criticism of the lock-free algorithm in Figure 1 is that it requires a strong synchronization primitive, namely `CAS2`. The fact that many lock-free object implementations are based on such primitives is no accident. Herlihy has shown that strong primitives are, in general, necessary for these implementations [7]. Nonetheless, Ramamurthy, Moir, and Anderson have shown that simple read and write instructions can be used to implement any strong synchronization primitive in a wait-free manner on a uniprocessor real-time system [18]. A *wait-free* object implementation must satisfy the following condition: if several tasks concurrently perform operations on the object, and if some proper subset of these tasks stop taking steps, then *each* of the remaining tasks

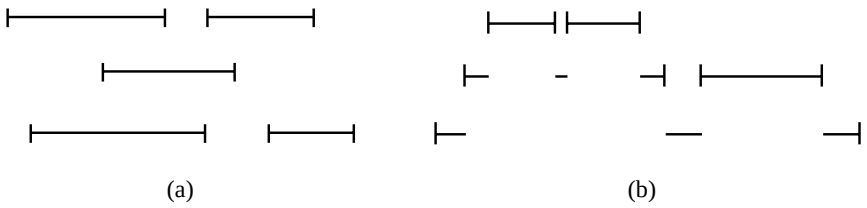

Figure 2 Line segments denote operations on shared objects with time running from left to right. Each level corresponds to operations by a different task. (a) Interleaved operations in an asynchronous multiprocessor system. Operations may overlap arbitrarily. (b) Interleaved operations in a uniprocessor real-time system. Two operations overlap only if one is contained within the other.

must complete its operation in a finite number of its own steps. This condition strengthens that required of lock-free implementations, and precludes waiting dependencies of any kind, including potentially unbounded retry loops.

The results of Ramamurthy et al. are based on the fact that certain task interleavings cannot occur in real-time systems. In particular, if a task τ_i performs an operation in the time interval $[t, t']$, and if another task τ_j performs an operation in the interval $[u, u']$, then it is not possible to have $t < u < t' < u'$, because the higher-priority task must finish its operation before relinquishing the processor. Requiring an object implementation to correctly deal with this interleaving is therefore pointless, because it cannot arise in practice. The distinction between traditional asynchronous systems, to which Herlihy's work is directed, and hard real-time systems is illustrated in Figure 2.

The results of [18] are based upon a task execution model like that depicted in Figure 2(b). This model is characterized by the following axioms.

Axiom 1: *Task τ_i may preempt task τ_j only if τ_i has higher priority than τ_j.* □

Axiom 2: *A task's priority can change over time, but not during any object access.* □

Axiom 1 is common to all priority-driven scheduling policies. Axiom 2 holds for most common policies, including RM [15], EDF [15], and DM scheduling [14]. The only common scheduling policy that we know of that violates Axiom 2 is least-laxity-first scheduling [16].

Most practical implementations of lock-free objects are based on compare-and-swap (`CAS`) and related primitives like load-linked/store-conditional (`LL/SC`) [8]. To enable such implementations to be used on systems that do not provide these primitives, Ramamurthy, Moir, and Anderson present two implementations of an object that supports `CAS`. (`LL/SC` can be implemented using `CAS` in constant time [1].) These implementations, which are summarized in the following theorems, use read/write and memory-to-memory `Move` instructions, respectively. `Move` is widely available on uniprocessors. For example, Intel's 80x86 and Pentium processors support the `Move` instruction. (In these theorems, N denotes the number of tasks that share an object.)

Theorem 3: *On any system satisfying Axioms 1 and 2,* `CAS` *can be implemented from reads and writes in a wait-free manner with $O(N)$ time and space complexity.* □

Theorem 4: *On any system satisfying Axioms 1 and 2,* `CAS` *can be implemented using* `Move` *in a wait-free manner with constant time and $O(N)$ space complexity.* □

3 LOCK-FREE TRANSACTIONS

In this section, we present an implementation of lock-free transactions on memory-resident data. We assume that transactions are invoked by a collection of prioritized tasks executing on the same processor. Our implementation is based on universal lock-free constructions by Anderson and Moir for implementing large objects and for implementing multi-object operations [1, 2]. The implementation uses a multi-word compare-and-swap (`MWCAS`) primitive for real-time systems proposed by Anderson and Ramamurthy [3].

3.1 Transaction Routines

Our transaction implementation, which is shown in Figure 3, consists of three procedures, `TR_Read`, `TR_Write`, and `TR_Exec`. These procedures support the "illusion" of a contiguous shared array *MEM* of memory words. In reality, the array is not stored in contiguous locations of memory, but is composed of a number of blocks. The `TR_Read` (`TR_Write`) procedure is invoked from user-supplied sequential transaction code to read words from (write words to)

```
type
    blktype = array[0..S − 1] of memwdtype;
    valtype = record blid: 0..B + NC − 1; ver: 0..V − 1 end;
    wdtype = record val: valtype; count: 0..B − 1; valid: boolean; pid: 0..N − 1 end
          /* The count, valid, and pid fields are used by the MWCAS/READ procedures. */

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 */
initially  (∀k : 0 ≤ k < B :: BANK[k] = ((NC + k, 0), 0, 0, true, 0)
                          ∧ BLK[NC + k] = (k^th block of initial value))
private variable
    copy: array[0..C − 1] of 0..B + NC − 1;       /* Indices for copy block of task τ_p */
    curr: array[0..B − 1] of valtype;     /* Task τ_p's current view of the MEM array */
    addrlist: array[0..B − 1] of pointer to wdtype;          /* Addresses for MWCAS */
    blklist: array[0..B − 1] of 0..B − 1;     /* List of blocks that have been accessed */
    oldval, newval: array[0..B − 1] of valtype;       /* Old and new values for MWCAS */
    dirty: array[0..B − 1] of 0..2;    /* 0 if block not accessed, 1 if read, 2 if modified */
    dcnt: 0..C − 1; done: boolean; i, j, numblks, blk: 0..B; tmp: 0..B + NC − 1;
    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)
procedure TR_Read(memwd: 0..BS − 1) returns memwdtype
1:  blk := memwd div S;
2:  if dirty[blk] = 0 then
3:     dirty[blk] := 1;
4:     curr[blk] := READ(&BANK[blk]);
5:     addrlist[numblks] := &BANK[blk];
6:     blklist[numblks] := blk;
7:     oldval[numblks] := curr[blk];
8:     numblks := numblks + 1
    fi;
9:  v := BLK[curr[blk].blid][memwd mod S];
10: if READ(&BANK[blk]) = curr[blk] then return v
11: else longjmp(env, 1)
    fi
```

Figure 3 Lock-free transaction implementation.

```
procedure TR_Write(memwd: 0..BS − 1;
                   value: memwdtype)
12:  blk := memwd div S;
13:  if dirty[blk] = 0 then
14:    curr[blk] := READ(&BANK[blk]);
15:    addrlist[numblks] := &BANK[blk];
16:    blklist[numblks] := blk;
17:    oldval[numblks] := curr[blk];
18:    numblks := numblks + 1
     fi;
19:  if dirty[blk] ≠ 2 then
20:    dirty[blk] := 2;
21:    memcpy(BLK[copy[dcnt]],
               BLK[curr[blk].blid],
               sizeof(blktype));
22:    if READ(&BANK[blk]) = curr[blk]
       then
23:      curr[blk].blid := copy[dcnt];
24:      dcnt := dcnt + 1
25:    else longjmp(env, 1)
       fi
     fi;
26:  tmp := curr[blk].blid;
27:  BLK[tmp][memwd mod S] := value
```

```
procedure TR_Exec(tr: function_ptr)
28:  done := false;
29:  while ¬done do
30:    dcnt, numblks := 0, 0;
31:    if setjmp(env) ≠ 1 then
32:      ∗tr();
33:      for j := 0 to numblks − 1 do
34:        i := blklist[j];
35:        newval[j] := curr[i];
36:        if dirty[i] = 2 then
37:          newval[j].ver :=
                 newval[j].ver + 1 mod V
           fi
         od;
38:      done := MWCAS(numblks, addrlist,
                       oldval, newval)
       fi;
39:    i := 0;
40:    for j := 0 to numblks − 1 do
41:      if done ∧ dirty[blklist[j]] = 2
         then
42:        copy[i] := oldval[blklist[j]].blid;
           i := i + 1
         fi;
43:      dirty[blklist[j]] := 0
       od
     od
```

Figure 3 *(continued)* Lock-free transaction implementation.

this array. The `TR_Exec` procedure takes this user-supplied code as input and executes it within the body of a lock-free retry loop. The input transaction is validated and committed in `TR_Exec` by invoking a `MWCAS` primitive, which is used in conjunction with an associated `READ` primitive. The semantics of `MWCAS` extends that of `CAS2` in Figure 1. We consider the implementation of the `MWCAS` and `READ` primitives in the next subsection.

When a transaction of task τ accesses a word in the implemented array of memory words, say $MEM[x]$, the block containing the x^{th} word is identified. If τ's

```
constant Boiler_temp = 0; Disp_temp = 1
/*  Locations 0 and 1 of the MEM array contain variables */
/*  Boiler_temp and Disp_temp, respectively.  */
private variable t : integer
procedure update_display()
    t := TR_Read(Boiler_temp);
    if TR_Read(Disp_temp) ≠ t then
        TR_Write(Disp_temp, t)
    fi
```

Figure 4 An example transaction.

transaction writes into $MEM[x]$, then τ must replace the corresponding block. The details of identifying blocks and replacing modified blocks are hidden from the programmer by means of the `TR_Read` and `TR_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 word of the MEM array. Thus, instead of writing "$MEM[1] := MEM[10]$", the programmer would write "`TR_Write`(1, `TR_Read`(10))". Figure 4 shows a simple example transaction, which updates the temperature display of a boiler. This transaction would be executed by calling `TR_Exec`(*update_display*).

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 array 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 using `MWCAS`.

In Figure 3, $BANK$ is a B-word shared array. Each element of $BANK$ contains a pointer to a block of size S and a version number (see below) for that pointer. 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. 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 τ_p's copy blocks. However, the roles of these blocks are not fixed. If τ_p successfully completes a transaction, then

τ_p reclaims the replaced blocks as copy blocks (lines 39-43). Thus, some of τ_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 `TR_Read` and `TR_Write` procedures. The `TR_Read` procedure first computes the index of the block containing the accessed word (line 1). If the block has not yet been read by this transaction, then it is marked as having been read (line 3), and is recorded in the transaction's *curr* array (line 4). This array gives the transaction's "current view" of *MEM*. The block index is also recorded in an array *blklist* (line 6), which is used later in reclaiming copy blocks when the transaction successfully completes. In addition, the address and old value of the block pointer are saved in arrays (lines 5 and 7) that are later used as parameters to the `MWCAS` procedure. The new value of the block pointer is determined later, prior to invoking `MWCAS` (lines 33-37). The `TR_Read` procedure completes by retrieving a value from the appropriate offset within the block that is accessed (line 9), and by performing a consistency check (lines 10 and 11), the purpose of which we describe below. The `TR_Write` procedure is similar to the `TR_Read` procedure, except that, when a block is first modified, it is recorded as having been modified, and a local copy of the block is made.

The *ver* counter associated with each block pointer in *BANK* records the current version number of the corresponding block. If a transaction successfully replaces a modified block, then it increments that block's version number. In contrast, if a block is read but not modified, then its block pointer and version number are not changed. This ensures that read-only transactions do not interfere with each other. A transaction can determine whether the i^{th} block has been changed by comparing the version number that it last read from $BANK[i]$ to the current version number of $BANK[i]$.

Before concluding this description, one subtlety that we have glossed over must be mentioned. If the *BANK* variable is modified by a transaction of task τ_q during the execution of a transaction of some lower-priority task τ_p, then τ_p may read inconsistent values from the *MEM* array. Because its `MWCAS` operation will subsequently fail, τ_p will not be able to install corrupted data. However, there is a risk that τ_p's sequential operation might cause an error, such as a division by zero or a range error. This problem is solved by ensuring that, if the version number of one of the blocks accessed by a transaction changes during that transaction, then control is returned from the `TR_Read` or `TR_Write` procedure to line 31 in `TR_Exec` using Unix-like *longjmp* calls. In this event, relevant data structures are reinitialized (lines 40-43) and the transaction is retried. Transactions can take advantage of this mechanism by re-reading previously

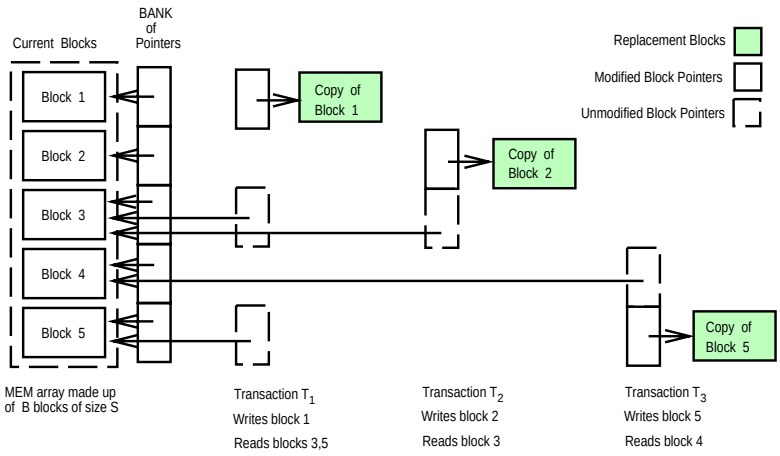

Figure 5 Implementation of the *MEM* array for lock-free transactions (depicted for $B = 5$).

accessed blocks in order to fail early in the event that such a block has been modified by another transaction.

In our implementation, read-only transactions do not interfere with one another, nor do transactions that modify disjoint sets of blocks. This is illustrated in Figure 5, which depicts three concurrent transactions T_1, T_2, and T_3. Transactions T_1 and T_2 do not interfere with each other because neither of them modifies a block accessed by both. However, T_3 can potentially interfere with T_1 because T_3 modifies block 5, which is read by transaction T_1.

The transaction implementation in Figure 3 can be optimized in several ways. For example, the code can be modified to support different block sizes. This would help to avoid false-sharing and fragmentation problems when using the *MEM* array to store a mixed collection of objects of different sizes. Also, objects such as queues can be incorporated into the implementation without using a copy-based solution.

The techniques shown here can potentially simplify transaction abort/recovery. In particular, each lock-free transaction execution has a distinct linearization step [11], and a transaction does not affect other transactions before it executes this step. Thus, complicated recovery procedures for undoing the effects of partially completed transactions are not required. (The ability to stop and restart a transaction arbitrarily would require some additional "clean up" code.

For example, if a transaction of a task were stopped while executing the loop at lines 40 through 43, then the next transaction of that task should finish the bookkeeping activities done in this loop.)

3.2 Implementing MWCAS and READ

We now show how to efficiently implement the **MWCAS** and **READ** primitives used in the previous subsection. Unfortunately, **MWCAS** is exceedingly difficult to implement efficiently in truly asynchronous systems [1, 6, 9, 22]. The most efficient known wait-free implementation [1] requires $\Theta(N^3M)$ time complexity to implement M words that can be accessed by N tasks. Fortunately, as shown by Anderson and Ramamurthy in [3], a W-word **MWCAS** can be implemented on a real-time uniprocessor in only $O(W)$ time (which is clearly optimal). In the remainder of this subsection, we present a brief overview of Anderson and Ramamurthy's implementation.

The code for implementing **MWCAS** and **READ** is shown in Figure 6. The implementation requires a **CAS** instruction. As mentioned in Section 2.2, **CAS** can be implemented on a real-time uniprocessor using a variety of other instructions. The words that may be accessed by the **MWCAS** and **READ** procedures are assumed to be of type *wdtype*. A word of this type consists of a *val* field, which contains an application-dependent value, and three fields that are used in the implementation, *count*, *valid*, and *pid*. In the transaction implementation of Section 3.1, the *val* field consists of a block pointer and version number.

We now explain how a **MWCAS** operation of task τ_r is executed. (In reading this description, it is important to keep in mind that Anderson and Ramamurthy's **MWCAS/READ** implementation is designed for real-time uniprocessors in which tasks are executed in accordance with the priority-based task model described in Section 2.2. In fact, if one assumes a conventional asynchronous task model, then the implementation does not work.) If **MWCAS** is invoked on a set of words, then it should either succeed and atomically change the current value of each word to the desired new value, or it should fail and change the value of no word. To understand how this is done, we need to specify how each word's "current value" is defined. Let w be a word of type *wdtype*. Then, the *current value* of w, denoted $cv(w)$, is defined as follows.

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

```
type                                  /* Assume N tasks, each MWCAS accesses at most B words */
  wdtype = record val: valtype; count: 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 */
  adtype = array[0..B-1] of pointer to wdtype;    /* Addresses to perform MWCAS on */
  vallist = array[0..B-1] of valtype              /* Lists of old and new values for MWCAS */
shared variable
  Status : array[0..N-1] of 0..2  initially 0;
  Save : array[0..N-1, 0..B-1] of valtype     /* Save word temporarily during MWCAS */
private variable                                  /* For task τ_p, where 0 ≤ p < N */
  in, as: array[0..B-1] of wdtype;      /* Values read and assigned to words by MWCAS */
  ovp: array[0..B-1] of boolean;        /* Indicates if a MWCAS operation is overlapped */
  i,j: 0..B; retval: boolean; wd: wdtype;   v: valtype

procedure MWCAS(nw : 0..B; ad: adtype;
        old, new: vallist) returns boolean /* MWCAS continued */
1:  Status[p], i := 0, 0;
2:  while i < nw ∧ Status[p] ≠ 1 do
3:    in[i], v, ovp[i] := *ad[i], *ad[i].val, false;
4:    if ¬in[i].valid ∧ Status[in[i].pid] ≠ 2
      then
5:      ovp[i] := true;
6:      v := Save[in[i].pid, in[i].count]
      fi;
7:    Save[p, i] := v;
8:    if old[i] ≠ v then Status[p] := 1
      else
9:      if old[i] ≠ new[i] ∧ ovp[i] then
10:       Status[in[i].pid] := 1
        fi;
11:     as[i] := (new[i], i, false, p);
12:     if ¬CAS(ad[i], in[i], as[i]) then
13:       Status[p] := 1
        fi;
14:     i := i + 1
      fi
    od;
15: retval := CAS(&Status[p], 0, 2);
16: for j := 0 to i - 1 do
17:   if retval ∧ old[j] ≠ new[j] then
18:     CAS(ad[j], as[j], (new[j], 0, true, p))
      else
19:     if ¬CAS(ad[j], as[j], in[j]) ∧ ovp[j]
        then
20:       Status[in[j].pid] := 1
        fi
      fi
    od;
21: return(retval)

procedure READ(ad: pointer to wdtype)
                              returns valtype
22: wd := *ad;
23: if wd.valid ∨ Status[wd.pid] = 2
    then
24:   return(wd.val)
    else
25:   return(Save[wd.pid, wd.count])
    fi
```

Figure 6 Wait-free implementation of MWCAS from CAS.

The shared array *Save* referred to in this definition is used to temporarily save the value of a word during a `MWCAS` operation on that word. The shared variable $Status[r]$ gives the "status" of task τ_r's latest `MWCAS` operation. $Status[r]$ is initialized to 0 when such an operation 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 8, 10, 13, and 20). A value of 2 in $Status[r]$ indicates that task τ_r's latest `MWCAS` operation has succeeded.

A `MWCAS` operation by task τ_r attempts to change the current values of the words it accesses in three phases. In the first phase (lines 1 through 14), τ_r attempts to modify fields within each of the (at most W) words it accesses; the k^{th} such word — call it w — is updated so that its *val* field contains the desired new value, the *count* 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 7). The *pid* and *count* fields of w are used by other tasks to retrieve the old value from the *Save* array (lines 6 and 25). Given the above definition of cv, it can be shown that the first phase of a `MWCAS` operation 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 task — then at the end of the 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 changing the value of $Status[r]$ from 0 to 2. By the definition of cv, this `CAS`, if successful, atomically changes the current value of each word that is accessed to the desired new value. The third and final phase consists of lines 16 through 21. In this phase, each word w that is accessed by the `MWCAS` operation of τ_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 task τ_r performs a subsequent `MWCAS` operation, reinitializing $Status[r]$ and modifying $Save[r, k]$ does not change the current value of any word.

Example. Figure 7 depicts the effects of a `MWCAS` operation m by task τ_4 on three words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. The values of relevant shared variables are shown at various points within this operation. 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 the variables after the first phase of m has completed, assuming no interferences by higher-priority tasks. The current value

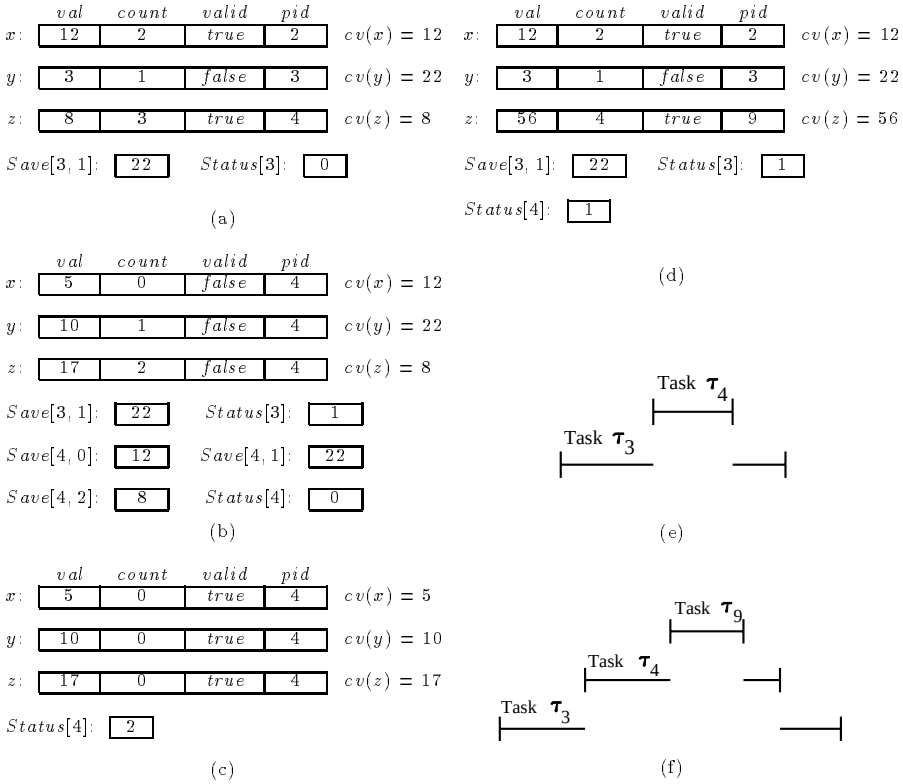

Figure 7 Task τ_4 performs a `MWCAS` operation on words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. The contents of relevant shared variables are shown (a) at the beginning of the operation; (b) after the loop in lines 3..17; (c) at the end of the operation, assuming success; and (d) at the end of the operation, assuming failure on word z. The operation interleaving that results in (c) is shown in (e) (τ_4 preempts τ_3). The operation interleaving that results in (d) is shown in (f) (τ_4 preempts τ_3, and τ_9 preempts τ_4).

of each word is unchanged. Also, *Status*[3] has been updated to indicate that task τ_3 (which must be of lower priority) has been interfered with. 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 tasks. 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). Inset (d) shows relevant variables at the termination of

m, assuming an interference on word z by task τ_9 (which must be of higher-priority) with new value 56. *Status*[4] is now 1, indicating the failure of τ_4's operation. *Status*[3] is still 1, indicating that τ_3's operation has also failed. Observe that τ_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. □

Having dispensed with the `MWCAS` procedure, the `READ` procedure can be readily explained. If the `READ` procedure is invoked with the address of word w as input, then it simply computes the current value of w.

Although the above description conveys the basic idea of the implementation, there are some subtleties that have been omitted for brevity. A more complete discussion that addresses these subtleties can be found in [3]. The results of this subsection yield the following theorem.

Theorem 5: *On any system satisfying Axioms 1 and 2,* `READ` *and* W*-word* `MWCAS` *operations can be implemented in a wait-free manner from* `CAS` *with* $O(1)$ *and* $O(W)$ *time complexity, respectively.* □

3.3 Simulation Results

Having considered how to implement and schedule tasks that perform lock-free transactions, we now briefly discuss some conclusions drawn from preliminary simulation experiments that compare lock-free, wait-free, and lock-based implementations of transactions on real-time uniprocessors. These experiments were conducted to determine achievable processor utilizations under the various transaction implementations that were tested. This was done by checking the schedulability of randomly generating task sets. A detailed description of the experimental methodology that was followed can be found in [3].

When comparing lock-free and lock-based implementations, the main conclusion to be drawn from the experiments is as follows: if lock-free loop costs are (on average) less than corresponding lock-based access costs (i.e., the cost of a lock/object-access/unlock sequence), then lock-free implementations perform better. Preliminary cost figures from actual implementations indicate that lock-free implementations of common data structures like queues, stacks, and linked lists are likely to be more efficient than lock-based implementations. On the other hand, lock-based implementations of more complex data structures like balanced trees are likely to be more efficient than lock-free ones. Also, in

practice, lock-free implementations may be preferable if most operations are read-only; the average loop cost in such situations is likely to be low. This is because, for many objects, read-only operations are inexpensive, because they do not entail any copying overhead.

The experiments also indicate that wait-free implementations perform better than their lock-free counterparts when access costs are identical. However, in practice, wait-free operation costs are typically much higher than corresponding lock-free costs, due to the additional overhead required to ensure wait-freedom. On the other hand, for transactions that only access very simple objects like read/write buffers, a wait-free implementation may be the best choice.

4 CONCLUDING REMARKS

The research outlined above leaves many opportunities for further work. Of foremost importance are definitive experimental studies that compare lock-free transactions with more conventional implementations. Further work is also needed on techniques for reducing the impact of failed loop tries and for minimizing state copying, particularly techniques that exploit our real-time task model. Finally, although we have targeted uniprocessor systems in which all data is memory-resident, it would be interesting to determine if some of the techniques we have proposed could be used in systems with disk-resident data or in systems with multiple processors.

Acknowledgements

We thank Steve Goddard for his valuable comments. The first three authors were 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. In addition, James Anderson was supported by an Alfred P. Sloan Research Fellowship, and Mark Moir was supported by a UNC Alumni Fellowship. Kevin Jeffay was supported by NSF grant CCR 9510156 and by grants from Intel and IBM.

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, 1995, pp. 168-182.

[3] J. Anderson and S. Ramamurthy, "A Framework for Implementing Objects and Scheduling Tasks in Lock-Free Real-Time Systems", to be presented at the 17th IEEE Real-Time Systems Symposium, December 1996. Available at the URL "`http://www.cs.unc.edu/~anderson/papers.html`".

[4] J. Anderson, S. Ramamurthy, and K. Jeffay "Real-Time Computing with Lock-Free Shared Objects", *Proceedings of the 16th IEEE Real-Time Systems Symposium*, 1995, pp. 28-37.

[5] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach", *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1992, pp. 127-132.

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

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

[8] 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.

[9] 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*, 1994, pp. 151-160.

[10] J. Harista, M. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints", *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990, pp. 331-343.

[11] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 463-492.

[12] T. Johnson and K. Harathi, "Interruptible Critical Sections", Technical Report TR94-007, Department of Computer Science, University of Florida, 1994.

[13] H. Kopetz and J. Reisinger, "The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem", *Proceedings of the IEEE Real-Time Systems Symposium*, 1993, pp. 131-137.

[14] J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks", *Performance Evaluation*, Vol. 2, No. 4, 1982, pp. 237-250.

[15] C. Liu and J. Layland, "Scheduling Algorithms for multiprogramming in a Hard Real–Time Environment", *Journal of the ACM*, Vol. 30, No. 1, 1973, pp. 46-61.

[16] A. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT Laboratory for Computer Science, 1983.

[17] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publications, 1991.

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

[19] P. Sorensen, *A Methodology for Real-Time System Development*, Ph.D. Thesis, University of Toronto, 1974.

[20] P. Sorensen and V. Hemacher, "A Real-Time System Design Methodology", INFOR, Vol. 13, No. 1, 1975, pp. 1-18.

[21] 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, 1990, pp. 1175-1185.

[22] N. Shavit and D. Touitou, "Software Transactional Memory", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204-213.