Lock-Free Transactions for Real-Time Systems*

James H. Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

We show that previous algorithmic and scheduling work concerning the use of lock-free objects in hard real-time systems can be extended to support real-time transactions on memory-resident data. Using our approach, transactions are not susceptible to priority inversion or deadlock, do not require complicated mechanisms for data-logging or for rolling back aborted transactions, and are implemented as library routines that require no special kernel support.

1 Introduction

In most real-time database systems, conventional mechanisms such as locks, timestamps, and serialization graphs are used for concurrency control. The main problem when using any of these mechanisms is that of handling conflicting operations. If an operation of a transaction creates a conflict, then one of two strategies may be used: either that transaction may be blocked, or one or more of the transactions involved in the conflict may be aborted. When using conflict resolution schemes that employ blocking, deadlock is a key issue, and mechanisms for deadlock avoidance or resolution are required. In contrast, deadlock is not a problem when using schemes that always resolve conflicts by aborting transactions. However, with such schemes, overhead associated with undoing the effects of partially completed transactions and then restarting them is a key issue.

Most existing conflict resolution schemes for real-time database systems are susceptible to priority inversion; a *priority inversion* occurs when a given transaction is blocked by another transaction of lower priority. Priority inversion is a problem even for systems that employ optimistic techniques, because in such systems, certain phases of a transaction (e.g., the validation phase or write phase) must be executed as critical sections [7, 8]. Priority inversion is often dealt with through the use of priority inheritance protocols or priority ceiling protocols [14]. These protocols dynamically adjust transaction priorities to ensure that a transaction within a critical section executes at a priority that is sufficiently high to bound the duration of any priority inversion. This functionality comes at the expense of additional overhead associated with maintaining information about transactions and the data they access.

In addition, in many database systems, major functional components are implemented as separate modules (e.g., transaction manager, lock manager, etc.), each of which consists of one or more processes. Transactions interact with these modules by invoking special primitives (e.g., *Begin_Transaction*, *End_Transaction*, *Read*, *Write*, *Commit*, *Abort*). Although structuring a system in this way is attractive from a software engineering standpoint, such an arrangement can result in significant interprocess communication overhead.

To summarize, although conventional concurrency control schemes provide a general framework for real-time transactions, actual implementations of these schemes can entail high overhead. In this paper, we propose a new approach for implementing real-time transactions on memory-resident data on uniprocessors. In our approach, transactions are implemented using a collection of library routines that hide from transaction programmers all details associated with managing concurrency. Transactions implemented using these routines are not susceptible to priority inversion or deadlock, and do not require complicated mechanisms for data-logging or for rolling back aborted transactions. In addition, they do not require special kernel support or additional processes for concurrency control; they therefore avoid interprocess communication overhead.

Our approach to implementing real-time transactions is based upon previous research involving the use of lock-free objects in real-time systems. An object implementation is *lock-free* iff it guarantees that, after a finite number of steps of any operation, *some* operation on the object completes. Lock-free objects

*The first three authors were supported, in part, by NSF contract CCR 9216421, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323. The third author was also supported by a UNC Alumni Fellowship. The fourth author was supported by grants from Intel and IBM.

```
type obj = record data: valtype; next: *obj end
shared variable queue_tail: *obj

procedure Enqueue(input: valtype)
local variable old_tail, new_tail: *obj

    *new_tail := (input, NULL);
    repeat old_tail := queue_tail
    until CAS2(queue_tail, old_tail ->next,
                    old_tail, NULL,
                    new_tail, new_tail)
```

Figure 1: Lock-free enqueue operation.

are usually implemented using "retry loops". Figure 1 depicts an example of a lock-free operation — an enqueue taken from a shared queue implementation given in [11]. In this example, an enqueue is performed by trying to thread an item onto the tail of the queue using a two-word compare-and-swap (`CAS2`) instruction.[1] This threading is attempted repeatedly until it succeeds. Note that the queue is never actually "locked".

Using lock-free object implementations, objects are optimistically accessed without locking, and an access is retried if an interference occurs. Operations are atomically validated and committed by invoking a strong synchronization primitive (like `CAS2`). In Figure 1, this validation step can fail for a task τ only if a higher-priority task performs a successful `CAS2` between the read of *queue_tail* by τ and the subsequent `CAS2` operation by τ.

From a real-time perspective, lock-free object implementations are of interest because they avoid priority inversions with no kernel 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 that can occur over any interval. The number of failed updates in such an interval is bounded by the number of task preemptions within that interval.

In this paper, we show that previous work on lock-free objects can be extended to apply to lock-free *transactions* on memory-resident databases. Our approach to implementing such transactions is to first implement a multi-word `CAS` primitive (`MWCAS`). This primitive can then be used as the basis for a lock-free retry loop in which operations on many objects are validated at once. Such an implementation differs from conventional optimistic concurrency control protocols in two respects. First, our implementations do not use locking. Therefore, they do not require special support for dealing with priority inversion. Second, our implementations allow transactions to be aborted at arbitrary points during their execution without the aid of recovery procedures. An important implication of this point is that data-logging is greatly simplified.

The rest of this paper is organized as follows. In Section 2, we review previous work involving the use of lock-free objects in real-time systems. Then, in Section 3, we outline our approach for implementing lock-free transactions. We end the paper with concluding remarks 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 it should be clear that 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, Ramamurthy, and Jeffay have shown that if tasks on a uniprocessor are scheduled appropriately, then such loops are indeed bounded [4]. We now explain intuitively why such bounds exist. For the sake of explanation, let us call an iteration of a retry loop a *successful update* if it successfully completes, and a *failed update* otherwise. Thus, a single 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 an uniprocessor, this can only happen if τ_i preempts the access of τ_j and then updates B successfully. However, τ_i preempts τ_j only if τ_i has higher priority than τ_j. Thus, at each priority level, 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. Using this information, it is possible to determine a bound on the number of failed updates in that interval. A set of tasks that share lock-free objects is schedulable if there is enough free processor time to accommodate the failed

[1] The first two parameters of `CAS2` are shared variables, the next two parameters are values to which the shared variables are compared, and the last two parameters are new values to be placed in the shared variables should both comparisons succeed.

updates that can occur over any interval.

In [4], scheduling conditions are established for *deadline-monotonic* (DM) [9] and *earliest-deadline-first* (EDF) [10] priority assignments. (The rate-monotonic (RM) [10] priority scheme is also considered, but the corresponding scheduling condition is a special case of that presented for the DM case.) 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 that $l_i = p_i$. Tasks are assumed to be sorted in nondecreasing order by their deadlines, 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, i.e., when there is no contention for the processor or for shared objects. Let s denote the execution time required 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 object update. Given this notation, the DM condition can be stated as follows.

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 every task τ_i, there exists some $t \in (0, l_i]$ such that*

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

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 object invocations (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 [1], we see that our condition essentially requires that the computation time of each task be "dilated" by the time it takes for one lock-free loop iteration.

We now turn our attention to the EDF scheme.

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*

$$\left(\sum_{j=1}^{N} \frac{c_j + s}{p_j}\right) \leq 1. \qquad \square$$

Informally, 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 [10] by requiring that the computation time of each task be dilated by the time it takes for 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 existing 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 not very tight — these bounds are based on the conservative assumption that each task interferes with all lower-priority tasks that execute concurrently. Scheduling conditions that incorporate tighter bounds will be given in a forthcoming paper. In these new conditions, tighter bounds are obtained by more accurately accounting for interferences that can occur between tasks. Also, the new conditions allow different lock-free accesses to have different loop costs. When implementing transactions, significant variations in loop costs are likely.

2.2 Hardware Support for Lock-Free Synchronization

A possible criticism of the lock-free algorithm given in Figure 1 is that it requires a strong synchronization primitive, namely `CAS2`. The fact that many lock-free object implementations are based on strong synchronization primitives is no accident. Herlihy has shown that such strong primitives are, in general, necessary for these implementations [5]. Herlihy's results are based upon a categorization of objects by "consensus number". An object has *consensus number* N if it can be used to solve N-process consensus, but not $(N+1)$-process consensus, in a wait-free[2] manner. Herlihy's results show that primitives with unbounded consensus numbers are necessary in general-purpose lock-free (or wait-free) object implementations. Such objects are called *universal* because they can be used to implement any other object. Herlihy's consensus-number

[2] *Wait-freedom* is a strong form of lock-freedom that precludes *all* waiting dependencies among tasks — including potentially unbounded operation retries.

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue
⋮	⋮
$2n-2$	n-register assignment
⋮	⋮
∞	memory-to-memory move and swap, compare&swap, load-linked/store-conditional

Figure 2: Herlihy's consensus-number hierarchy.

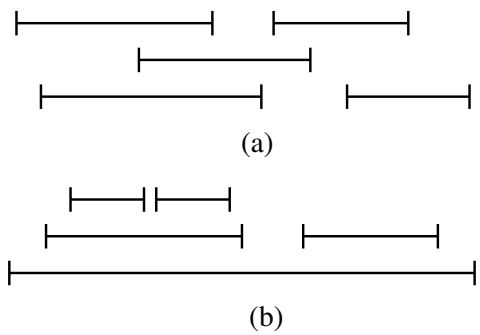

Figure 3: Line segments denote operations on shared objects with time running from left to right. Each level corresponds to operations by a different task. (a) Interleaved operations in a asynchronous system. Operations may overlap arbitrarily. (b) Interleaved operations in a uniprocessor real-time system. Two operations overlap only if one is contained within the other.

hierarchy is shown in Figure 2.

Although Herlihy's hierarchy is of fundamental importance for truly asynchronous systems, Ramamurthy, Moir, and Anderson recently showed that, for uniprocessor-based real-time systems, Herlihy's hierarchy collapses, i.e., reads and writes can be used to implement any lock-free object [13]. The basis for this result is the realization that certain task interleavings cannot occur within real-time systems. In particular, if a task τ_i accesses an object in the time interval $[t, t']$, and if another task τ_j accesses that object 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 3.

The results of [13] are based upon an execution

```
shared var Fin, Prp : valtype ∪ ⊥
private var v : valtype
initially Fin = ⊥ ∧ Prp = ⊥

procedure decide(val : valtype) returns valtype
1 : if Fin = ⊥ then
2 :     if Prp = ⊥ then
3 :         Prp := val
        fi;
4 :     if Fin = ⊥ then
5 :         v := Prp;
6 :         Fin := v
        fi
    fi;
7 : return Fin
```

Figure 4: Consensus using reads and writes.

model like that depicted in Figure 3(b). Two key assumptions underlie this model: (i) a task τ_i may preempt another task τ_j only if τ_i has higher priority than τ_j; (ii) a task's priority can change over time, but not during any object access. Assumption (i) is common to all priority-driven scheduling policies. Assumption (ii) holds for most common policies, including RM, DM, and EDF scheduling. The only common scheduling policy that we know of that violates assumption (ii) is least-laxity-first (LLF) scheduling [12].

The collapse of Herlihy's hierarchy for hard real-time uniprocessor systems is established in [13] by giving a wait-free (and hence lock-free) algorithm that solves the consensus problem in such systems using only reads and writes. This algorithm is shown in Figure 4. In this algorithm, each task chooses a decision value by invoking the procedure *decide*.

Procedure *decide* uses two shared variables, *Prp* ("propose") and *Fin* ("final"). Each task that does not detect the input of another task in *Prp* or *Fin* writes its own value into *Prp*. Having ensured that some value has been proposed (lines 1 to 3), a task copies the proposed value to *Fin*, if necessary (lines 4 to 6). It is easy to see that no task returns before some task's input value is written into *Fin*, and that all tasks return a value read from *Fin*. It can be further shown that the first value written into *Prp* is the only value written into *Fin*. The correctness of the algorithm easily follows from this fact, yielding the following theorem. (It is easy to show that the algorithm of Figure 4 does *not* correctly solve the consensus problem in a truly asynchronous system consisting of two or more processes. Thus, it does not contradict the fact the reads and writes have consensus number 1 for such systems.)

Theorem 3: *Consensus can be implemented with constant time and space using reads and writes on a hard*

real-time uniprocessor system. □

Given an implementation of consensus objects, any shared object can be implemented in a lock-free manner [5]. However, such implementations usually entail high overhead. More practical lock-free implementations are based on universal primitives such as `CAS` and load-linked/store-conditional (`LL/SC`)[6]. To enable such implementations to be used in real-time uniprocessor systems, Ramamurthy, Moir, and Anderson present two implementations of an object that supports `Read` and `CAS`. (`LL/SC` can be implemented using `Read` and `CAS` in constant time [2].) These implementations, which are summarized in the following theorems, use read/write and memory-to-memory `Move` instructions, respectively. Although `Move` is rare in multiprocessors, it is widely available on uniprocessors. For example, uniprocessor systems based on Intel's 80x86 processor and the Pentium line of processors support the move instruction. (In these theorems, N denotes the number of tasks that share an object.)

Theorem 4: `Read` *and* `CAS` *can be implemented in a wait-free manner on a hard real-time uniprocessor system with $O(N)$ time and space complexity using reads and writes.* □

Theorem 5: `Read` *and* `CAS` *can be implemented in a wait-free manner on a hard real-time uniprocessor system with constant time and $O(N)$ space complexity using* `Move`*.* □

3 Lock-Free Transactions

In this section, we outline 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 mostly based on the universal lock-free constructions of large objects and multiple objects by Anderson and Moir [2, 3].

In contrast to conventional schemes for concurrency control, when lock-free algorithms are used, transactions are executed as if they do not access any shared data — i.e., such transactions can be viewed as being independent. As a result, mechanisms are not needed to handle priority inversion, deadlock, or data conflicts, or to undo the effects of partially-completed transactions that are aborted.

In our lock-free implementation, which is shown in Figure 5, a transaction commits and validates at the same time by performing a `MWCAS` operation.[3] A transaction successfully completes, and the corresponding modifications to data take effect, only if the `MWCAS` operation succeeds. Thus, transactions can be aborted arbitrarily without executing expensive recovery procedures. This can be advantageous in situations that require transactions with firm deadlines to be aborted due to transient overload conditions. In such situations, transactions can be arbitrarily terminated by the system without fear of compromising data consistency.

One possible criticism of our implementation is that the `MWCAS` primitive is not supported by the hardware in most systems. However, Anderson and Moir have shown that `MWCAS` can be implemented using the single-word `CAS` primitive in general asynchronous systems [3]. Furthermore, because real-time systems only allow a subset of the transaction interleavings in asynchronous systems, these constructions can be simplified for real-time systems using techniques similar to those presented in Section 2.2.

In our lock-free construction, all data is stored in a single array. However, we do not require the array to be stored in contiguous memory locations. Instead, we provide the "illusion" of a contiguous array. Before describing the code in Figure 5, we explain how this is achieved, and how the details of the implementation are hidden from the programmer.

The implemented array *MEM* is partitioned into B blocks of size S. (Figure 6 depicts this arrangement for $B = 5$.) The first block contains array locations 0 through $S - 1$, the second block contains locations S through $2S-1$, and so on. (We assume that blocks have the same size only because it simplifies the presentation of the ideas in the paper.) A bank of pointers — one for each block — is used to point to the blocks that make up the array. In order to modify the contents of the array, a task makes a copy of each block to be changed, and then attempts to update the bank of pointers so that they point to the modified blocks.

When a transaction of task p accesses a word in the array, say $MEM[x]$, the block containing the xth word is identified. If p's transaction writes into $MEM[x]$, then p 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 record-keeping. These routines are called by the transaction in order to read or write an element of the *MEM* array. Thus, instead of writing

[3] The `MWCAS` procedure takes four input parameters. The first parameter specifies the number of words on which a `CAS` operation is to be performed. The remaining parameters specify the words' addresses, old values, and new values, respectively. The operation returns true if a `CAS` operation can be successfully performed on all the words simultaneously, and returns false otherwise.

```
type blktype = array[0..S − 1] of wordtype;  ptrtype = record addr: 0..B + NT − 1;  ver: integer end

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

private variable  addrlist, copy: array[0..T − 1] of 0..B + NT − 1;  oldlst, ptrs: array[0..B − 1] of ptrtype;
                  dirty, touch: array[0..B − 1] of boolean;  src, dest: *blktype;  nw, dirtycnt: 0..T;  i, blkidx: 0..B − 1;
                  oldval, newval: array[0..B − 1] of ptrtype;  blklist: sorted_list of 0..B − 1;  env: jmp_buf
initially (∀k : 0 ≤ k < T :: copy[k] = pT + k) ∧ (∀k : 0 ≤ k < B : ¬touch[k] ∧ ¬dirty[k])

procedure Read(address: 0..BS − 1) returns wordtype
1:  blkidx := address div S;
2:  if ¬touch[blkidx] then touch[blkidx] := true;  blklist.insert(blkidx);  ptrs[blkidx] := BANK[blkidx] fi;
3:  v := BLK[ptrs[blkidx]][address mod S];
4:  if BANK[blkidx].ver = ptrs[blkidx].ver then return v else longjmp(env, 1) fi

procedure Write(address: 0..BS − 1; val: wordtype)
5:  blkidx := address div S;
6:  if ¬touch[blkidx] then
        touch[blkidx] := true;
        blklist.insert(blkidx);
        ptrs[blkidx] := BANK[blkidx]
    fi;
    if ¬dirty[blkidx] then
        dirty[blkidx] := true;
7:      src := BLK[ptrs[blkidx].addr];
8:      dest := BLK[copy[dirtycnt]];
        memcpy(dest, src, sizeof(blktype));
9:      if BANK[blkidx].ver = ptrs[blkidx].ver then
            oldlst[blkidx] := ptrs[blkidx];
            ptrs[blkidx].addr := copy[dirtycnt];
            dirtycnt := dirtycnt + 1
        else longjmp(env, 1)
        fi
    fi;
10: BLK[ptrs[blkidx].addr][address mod S] := val

procedure LF_Transaction(tr: function_pointer)
    while true do
        dirtycnt, nw := 0, 0;
11:     if setjmp(env) ≠ 1 then *tr() fi;
12:     while ¬blklist.empty() do
            i := blklist.delete_list_head();
            addrlist[nw] := i;
            if dirty[i] then
                oldval[nw] := oldlst[i];
                newval[nw] := (ptrs[i].addr, ptrs[i].ver + 1);
                dirty[i] := false
            else oldval[nw], newval[nw] := ptrs[i], ptrs[i]
            fi;
            touch[i], nw := false, nw + 1
        od;
13:     if MWCAS(nw, addrlist, oldval, newval) then
            for i := 0 to dirtycnt − 1 do copy[i] := oldlst[i].addr od;
            return
        fi
    od
```

Figure 5: Implementation of lock-free transactions (*LF_Transaction*) and the associated *Read* and *Write* procedures.

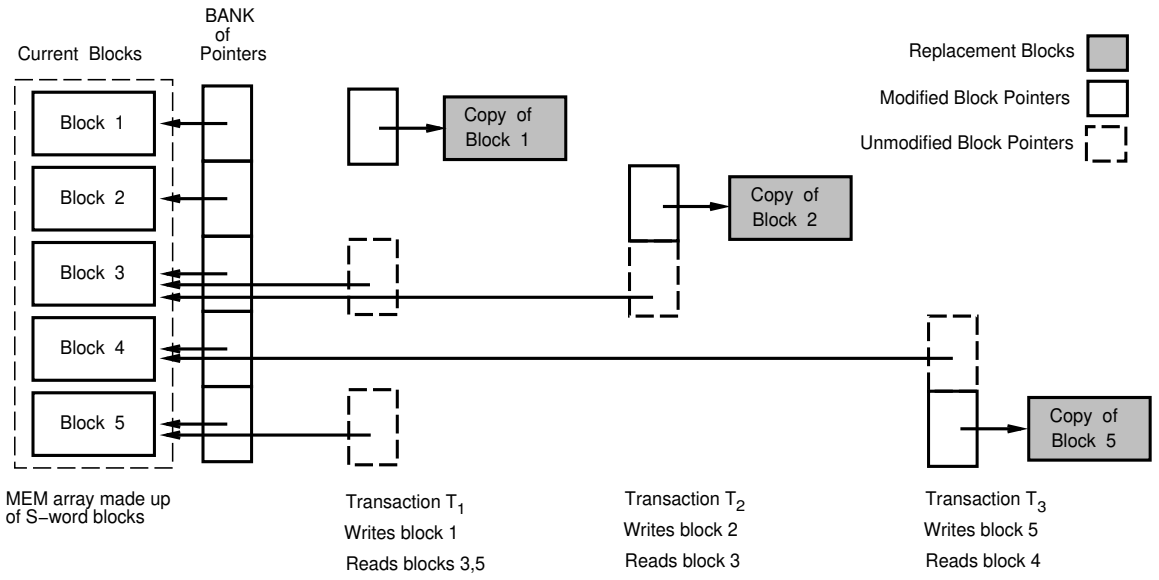

Figure 6: Implementation of the *MEM* array for lock-free transactions (depicted for $B = 5$).

"$MEM[1] := MEM[10]$", the programmer would write "$Write(1, Read(10))$".

In Figure 5, $BANK$ is a B-word shared variable. Each element of $BANK$ contains a pointer to a block of size S and a version number for the pointer. (Actually, each pointer is an index into an array of blocks BLK.) The B blocks pointed to by $BANK$ constitute the current value of the array MEM. We assume that an upper bound T is known on the number of blocks modified by any transaction. Because a task's transaction copies a block before modifying it, T "copy" blocks are required per task. Therefore, a total of $B + NT$ blocks are used. These blocks are stored in the array BLK. Although blocks $BLK[NT]$ to $BLK[NT + B - 1]$ are the initial blocks of the array, and $BLK[pT]$ to $BLK[(p+1)T - 1]$ are task p's initial copy blocks, the roles of these blocks are not fixed. If p successfully completes a transaction by performing a `MWCAS` operation, then p reclaims the replaced blocks as copy blocks. Thus, the copy blocks of one task may become blocks of the array MEM, and then later become copy blocks of another task.

Task p performs a lock-free transaction by calling the *LF_Transaction* procedure, which repeatedly performs the transaction until it executes a successful `MWCAS` operation. The user-supplied transaction code accesses the MEM array in a sequential manner using the *Read* and *Write* procedures. The *Read* procedure first computes the index of the block containing the accessed word, and then marks the block as "touched". Before returning the value from the appropriate offset within that block, the block index is inserted into a sorted list *blklist*, and the block pointer is recorded in *ptrs*.

Like the *Read* procedure, the *Write* procedure first computes the block to be accessed and records it as "touched", if necessary. Then, if *Write* is modifying the block for the first time during the transaction, the block is copied into one of the task's copy blocks, and the block is marked as "dirty". The copy block is then made part of the local version of MEM by linking it into *ptrs*. Finally, the displaced block is recorded in *oldlst* for possible reclaiming later, and the appropriate word is modified in the local copy of the block.

The *ver* counter[4] associated with each block pointer in $BANK$ records the block's current "version" number. If a transaction successfully replaces a modified block, then it increments the block's version number. Thus, a transaction determines whether the ith block has been changed by comparing the version number that it last read from $BANK[i]$ to the current version number of $BANK[i]$. Note that if a successful transaction reads a block but does not modify it, then the block is marked as "touched" but not "dirty", and the subsequent successful `MWCAS` does not change the block pointer or version number.

A complication arises in our implementation when the $BANK$ variable is modified by a higher-priority task's transaction during the execution of task p's transaction, thereby causing p to read inconsistent values from the MEM array. Because its `MWCAS` operation will subsequently fail, p will not be able to install corrupted data. However, there is a risk that 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 task p detects that the version number of one of the blocks accessed by it has been modified since its most recent access, then control is returned from the *Read* or *Write* procedure to line 12 in *LF_Transaction* using Unix-like `setjmp` and `longjmp` system calls. Task p then "cleans up" by reinitializing *blklist*, *dirty*, and *touch*, fails the `MWCAS` operation, and retries the transaction. Transactions can take advantage of this mechanism by re-reading previously accessed blocks in order to fail early in the event that the block has been modified by another transaction.

An example transaction that updates the temperature display of a boiler is given in Figure 7. Under our implementation, the transaction is executed by calling *LF_Transaction*(*update_display*). As can be seen, transactions implemented under conventional schemes can be easily modified to work under our lock-free scheme.

In our implementation, concurrent read operations do not interfere with one another. Also, concurrent transactions that modify disjoint sets of blocks do not interfere with one another, as illustrated in Figure 6. The figure 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.

Observe that the scheduling conditions presented in Section 2.2 apply to task sets. These conditions can be applied directly to periodic (or sporadic) transactions even if the objects accessed by a transaction are not known in advance, i.e., a transaction can access data anywhere in the MEM array. In many applications, it should be possible to tighten these scheduling conditions by more carefully accounting for the kinds of conflicts that can arise among transactions. Note that a transaction T_i can interfere with another transaction T_j only if T_i has higher priority than T_j and T_i writes a block that is accessed by T_j.

[4] Our implementation uses unbounded counters to implement block version numbers. In a forthcoming paper, we show that these counters can be bounded.

```
procedure update_display()
local variable t: integer
  t := Read(Boiler_temp);
  if Read(Disp_temp) ≠ t then Write(Disp_temp, t) fi
```

Figure 7: An example Transaction.

In some applications, it is essential to back up the database in stable storage. This is usually achieved by logging operations on the database. The techniques presented here have the potential to greatly simplify transaction logging, mainly because they do not require procedures for recovery from aborted transactions. When a task performs a transaction, it can update the database and the log file simultaneously (treating the log file as simply another block of memory to update). Because all transactions access the log file, a simplistic application of this approach would result in each transaction interfering with all lower-priority concurrent transactions, which would adversely impact schedulability. Log-file updates should therefore be handled differently from other memory accesses. In particular, if a transaction fails to update the log file, then it should retry only the log file update (using a very short retry loop like that in Figure 1), as opposed to retrying the entire transaction.

4 Concluding Remarks

The research outlined above leaves many opportunities for further research. Of foremost importance are experimental studies that compare lock-free transactions with more conventional implementations. It would also be interesting to determine if lock-free algorithms could be used in systems with disk-resident data. Finally, it would be interesting to investigate the applicability of these techniques within multiprocessors and distributed systems.

Acknowledgement: We thank Steve Goddard and the anonymous referees for their valuable comments.

References

[1] 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*, Oxford, UK, 1992, pp. 127-132.

[2] 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.

[3] 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.

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

[6] 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.

[7] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proceedings of the Seventeenth International Conference on Very Large Databases*, 1991, pp. 35-46.

[8] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6, No. 2, 1981, pp. 213-226.

[9] 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.

[10] 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.

[11] H. Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. Dissertation, Columbia University, 1992.

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

[13] 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*, to appear.

[14] 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.