To be presented at the 17th IEEE Real-Time Systems Symposium, December 1996.

A Framework for Implementing Objects and Scheduling Tasks in Lock-Free Real-Time Systems*

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

Abstract

We present an integrated framework for developing real-time systems in which lock-free algorithms are employed to implement shared objects. There are two key objectives of our work. The first is to enable functionality for object sharing in lock-free real-time systems that is comparable to that in lock-based systems. Our main contribution toward this objective is an efficient approach for implementing multi-object lock-free operations and transactions. A second key objective of our work is to improve upon previously proposed scheduling conditions for tasks that share lock-free objects. When developing such conditions, the key issue is to bound the cost of operation "interferences". We present a general approach for doing this, based on linear programming.

1. Introduction

Most work on implementing shared objects in preemptive real-time uniprocessor systems has focused on using critical sections to ensure object consistency. The main problem that arises when using critical sections is that of priority inversion. A *priority inversion* exists when a given task must wait on a task of lower priority to release a critical section. A number of schemes have been proposed to bound blocking times associated with priority inversion; these include the priority inheritance protocol [15, 17], the priority ceiling protocol (PCP) [4, 15, 17], the dynamic PCP (DPCP) [7], and the earliest-deadline-first scheme with dynamic deadline modification (EDF/DDM) [10].

An alternative to such schemes is to use *lock-free* algorithms for implementing objects. The general utility of lock-free objects in real-time applications was first established by Anderson, Ramamurthy, and Jeffay [3]. Operations on lock-free objects are usually implemented using "retry loops".[1] Figure 1 depicts lock-free queue operations that are implemented in this way. An item is enqueued in this implementation by using a two-word compare-and-swap (CAS2) instruction[2] to atomically update a shared tail pointer and the "next" pointer of the last item in the queue. The CAS2 instruction is attempted repeatedly until it succeeds. Dequeue is implemented similarly. Note that the queue is never "locked" by any task.

On the surface, it is not immediately apparent that lock-free shared objects can be employed if tasks must adhere to strict timing constraints. In particular, lock-free operations may interfere with each other, and repeated interferences can cause a given operation to take an arbitrarily long time to complete. For example, the enqueue implementation in Figure 1 allows a given task to experience repeated failed loop iterations due to "interfering" concurrent enqueues. Anderson et al. observed that, on a uniprocessor, the cost of such failed loop iterations over an interval of time can be bounded by the number of job releases[3] within that interval [3]. This observation is the basis of scheduling conditions presented in [3] for the rate-monotonic (RM) [13], deadline-monotonic (DM) [12], and earliest-deadline-first (EDF) [13] schemes. Note that, with lock-free objects, the overhead term is the cost of failed loop iterations, and lower-priority tasks are most likely to pay this overhead cost. In contrast, when two tasks access a common object under a lock-based scheme, the higher-priority task pays the price of ensuring object consistency, because it is the higher-priority task that can be blocked by a priority inversion.

Though encouraging, the results of Anderson et al. can only be viewed as a first step towards a general framework for lock-free object sharing in real-time systems. For example, in bounding failed loop iterations, they assume that all retry loops are of uniform cost, and that each job release

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

[1] In this paper, we consider both lock-free objects and the related notion of a "wait-free" object. Formally, if several processes attempt to access a *lock-free* (*wait-free*) object concurrently, and if some proper subset of these processes stop taking steps, then one (each) of the remaining processes completes its access in a finite number of its own steps. The stronger wait-free condition precludes waiting dependencies of any kind, including potentially unbounded retry loops.

[2] 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 be assigned to the variables if both comparisons succeed.

[3] We use the term *job* to refer to a single task invocation; a job is *released* when it becomes available for execution.

```
type Qtype = record data: valtype; next: *Qtype end
shared variable Head, 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)

procedure Dequeue() returns *Qtype
    repeat old := Head;
            if old = NULL then return NULL fi;
            new := old->next
    until CAS2(&Head, &(old->next), old, new, new, NULL);
    return(old)
```

Figure 1. Lock-free queue implementation.

causes an interference. Such assumptions lead to scheduling conditions that are somewhat pessimistic. In addition, Anderson et al. only considered operations on single objects. This stands in contrast to prior work on lock-based object sharing, where "nested" critical sections have been considered in which multiple objects are accessed.

In this paper, we present an integrated framework for implementing lock-free objects in real-time systems, and for scheduling tasks that share such objects. This framework resolves all of the shortcomings noted in the previous paragraph. We first consider the problem of implementing multi-object lock-free operations and transactions, the lock-free counterpart to nested critical sections. It has been shown that such implementations can be developed by using a multi-word compare-and-swap (MWCAS) primitive within a lock-free retry loop in which many objects are accessed at once; the semantics of MWCAS generalizes that of CAS2 used in Figure 1. A general MWCAS primitive is impractical to provide in hardware, so it must be implemented in software. For our purposes, such an implementation should be lock-free or wait-free. Unfortunately, previous lock-free and wait-free implementations of primitives like MWCAS have rather high worst-case time complexity [1, 5, 9, 18]. Thus, they are of limited utility in real-time systems.

One of the main contributions of this paper is to show that a wait-free MWCAS primitive *can* be implemented efficiently if one assumes a priority-based uniprocessor task scheduler. Our implementation of MWCAS was inspired by recent results of Ramamurthy, Moir, and Anderson, who were the first to realize that properties of priority-based schedulers can be exploited to simplify wait-free (and lock-free) algorithms [16]. The basis of this realization is the fact that certain task interleavings cannot occur when using such schedulers. In particular, if a task T_i accesses an object in the time interval $[t, t']$, and if another task T_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. In effect, accesses of high-priority tasks appear atomic to low-priority tasks.

The second major contribution of this paper is a general approach, based on linear programming, for obtaining an accurate bound on the cost of operation interferences. In this approach, the total cost of interferences in T_i and higher-priority tasks over an interval $\mathcal{I}$ is first expressed as a linear expression involving a set of variables. Each variable in the expression represents the number of interferences in each lock-free operation due to higher-priority jobs 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 T_j in $\mathcal{I}$ is bounded by the number of job releases of T_j in $\mathcal{I}$. Finally, an upper bound on the total cost of interferences in T_i and higher-priority tasks during $\mathcal{I}$ is calculated using linear programming. We show that this approach can be applied to most common scheduling schemes. The scheduling conditions we derive are much tighter than those originally reported in [3].

How do lock-free objects implemented using the results of this paper compare to lock-based implementations? To answer this question, we conducted a number of simulation experiments involving randomly generated sets of tasks that perform both single- and multi-object operations. Although it is hard to draw definitive conclusions about specific applications from these experiments, the results suggest that lock-free implementations are probably more efficient if, on average, the cost of a retry-loop is less than that of a corresponding lock/object-access/unlock sequence. Thus, for example, lock-free implementations may be preferable if a high percentage of operations are read-only (such operations can usually be implemented with inexpensive retry loops), or if lock and unlock primitives are costly.

The rest of this paper is organized as follows. In Section 2, we present our wait-free implementation of the MWCAS primitive, and briefly consider some applications of it. We then present our approach to task scheduling in Section 3, and discuss results from simulation experiments in Section 4. We end the paper with concluding remarks in Section 5.

2. The MWCAS Primitive

MWCAS is a useful primitive for two reasons. First, it simplifies the implementation of many lock-free objects; queues, for instance, are easy to implement with MWCAS, but harder to implement with single-word primitives. Second, it can be used to implement multi-object operations and transactions. For example, an operation that dequeues an item off of one queue and enqueues it onto another could be implemented by using MWCAS to update both queues.

2.1. Lock-Free Transactions

The idea of using MWCAS to atomically access many objects can be generalized to implement arbitrary lock-free transactions on memory-resident data. Such an implementation was presented recently by Anderson, Ramamurthy, Moir, and Jeffay [2]. In this implementation, memory that can be read and written by transactions is partitioned into blocks of words. These blocks are accessible by means of a bank of pointers, one for each block. In order to write a word, a transaction makes a copy of each block to be changed, and then attempts to replace the old version of that block by its modified copy (using MWCAS). Blocks that are read but not modified are not copied (thus, read-only transactions are executed with low overhead). A "version counter" is associated with each block pointer. A transaction attempts to validate and commit by invoking MWCAS to atomically *compare* version counters of accessed blocks and to *swap* in new pointers and version counters for modified blocks. If this MWCAS invocation is unsuccessful, then the transaction is retried. This transaction implementation has the desirable property that read-only transactions do not interfere with each other. This is because pointers and version counters are updated only for modified blocks.

The transaction implementation just described shows that functionality comparable to that in lock-based systems can be achieved in lock-free systems by using a MWCAS primitive. In the following subsection, we consider the question of how to implement such a primitive efficiently.

2.2. A Wait-Free Implementation of MWCAS

Figure 2 depicts our implementation of MWCAS and an associated Read primitive. The implementation requires a CAS instruction. Requiring CAS is not a severe limitation, because Ramamurthy et al. [16] have shown that CAS can be implemented on most priority-based real-time systems from reads and writes with time complexity that is linear in the number of tasks sharing a common object. Many processors, in fact, either provide CAS directly or provide synchronization instructions that can be used to implement CAS in constant time. For example, CAS is provided in hardware on the Motorola 680x0 line of processors and on the Intel Pentium. It can be implemented in constant time on the Intel 80x86 line of processors using a memory-to-memory move instruction (see [16] for details), and on the PowerPC using *load-linked* and *store-conditional* instructions.

In our implementation, a task performs a MWCAS operation on a collection of words by invoking the MWCAS procedure. This procedure takes as input an integer parameter indicating the number of words to be accessed, an array containing the addresses of the words to be accessed, and arrays containing old and new values for these words. We assume that task identifiers range over $\{0, \ldots, N-1\}$ and that each MWCAS operation accesses at most B words. A task reads a word by invoking the Read procedure, which takes as input the word's address. The words that may be accessed by the MWCAS and Read procedures are assumed to be of type *wordtype*. 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* ($\lceil \log B \rceil$ bits), *valid* (one bit), and *pid* ($\lceil \log N \rceil$ bits). In most applications, the *val* field would contain an object pointer, and perhaps a small amount of control information. For example, in the implementation of lock-free transactions discussed above, a version counter is stored as control information.

We present below a detailed description of the MWCAS and Read procedures. In reading this description, it is important to keep in mind that these procedures were designed assuming a priority-based uniprocessor task model. In fact, if one assumes a conventional asynchronous task model, then the implementation does not work. The priority-based task model assumed here is the same as that considered in the work of Ramamurthy et al. [16]. This model is characterized by two simple requirements: (i) a task's priority may change over time, but not during a MWCAS or Read operation; (ii) if a given task has an enabled statement at a state, then no lower-priority task has an enabled statement at that state. Note that these requirements imply that if a given task begins executing one of the procedures in Figure 2, then no lower-priority task may execute any statement until that procedure invocation completes.[4]

With this task model in mind, we now explain how the implementation works. We begin with an overview of the MWCAS procedure. We follow this by an example that illustrates the key ideas. After this, we present a brief overview of the (much simpler) Read procedure. We then conclude by considering several subtleties of the implementation that are not addressed in our initial overview.

We explain the MWCAS procedure by focusing on a MWCAS operation by task T_r. Such an operation is executed in three phases. In the first phase (lines 1 through 17), the k^{th} word that is accessed by T_r — 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 14 and 15). In addition, the old value of w is saved in the shared variable $Save[r, k]$ (line 10). The *pid* and *count* fields of w are used by other tasks to retrieve the old value from the *Save* array. The *pid* field is also used as an index into the *Status* array, the role of which is described below.

To understand the "effect" the first phase has on the words that are accessed, it is necessary to understand how each word's "current value" is defined.

Definition 1: Let w denote a variable of type *wordtype* that is accessible by a MWCAS or Read operation. Then, the *current value* of w, denoted $Val(w)$, is defined as follows.

[4] The only common scheduling policy that we know of that violates these requirements is least-laxity-first (LLF) scheduling [14]. Under LLF scheduling, the priority of a task invocation can change during its execution.

```
type                                                    /* Assume N tasks, each MWCAS accesses at most B words */
    wordtype = 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; the valid field should be initially true */

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

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

private variable                                                        /* For task T_p, where 0 ≤ p < N */
    init, assn: array[0..B − 1] of wordtype;                            /* Values initially read and assigned to words by MWCAS */
    ovlap: array[0..B − 1] of boolean;                                  /* Indicates if a lower-priority MWCAS operation is overlapped */
    i,j: 0..B + 1; retval: boolean; word: wordtype; val: valtype

procedure MWCAS(numwds: 0..B; addr: addrlisttype;
                old, new: vallisttype) returns boolean
1:  Status[p] := 0;
2:  i := 0;
3:  while i < numwds ∧ Status[p] ≠ 1 do
4:      init[i] := *addr[i];
5:      if init[i].valid ∨ Status[init[i].pid] = 2 then
6:          ovlap[i] := false;
7:          val := init[i].val
        else
8:          ovlap[i] := true;
9:          val := Save[init[i].pid, init[i].count]
        fi;
10:     Save[p, i] := val;
11:     if old[i] ≠ val then Status[p] := 1
        else
12:         if old[i] ≠ new[i] ∧ ovlap[i] then 13: Status[init[i].pid] := 1 fi;
14:         assn[i] := (new[i], i, false, p);
15:         if ¬ CAS(addr[i], init[i], assn[i]) then 16: Status[p] := 1 fi;
17:         i := i + 1
        fi
    od;

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

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

Figure 2. Wait-free implementation of MWCAS from CAS.

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

We see from this definition that $Val(w)$ depends on the value of $Status[r]$ if $w.pid = r$. $Status[r]$ is initialized to 0 when a MWCAS operation of T_r begins (line 1). If the operation is interfered with by other MWCAS operations, or if the current value of some word accessed by the operation differs from the old value specified for that word, then $Status[r]$ is assigned the value 1 (lines 11, 13, 16, and 23). A value of 2 in $Status[r]$ indicates that task T_r's latest MWCAS operation has succeeded.

With Definition 1 in mind, the "effect" of the first phase of a MWCAS operation can now be understood. This phase does not change the current value of any word that is accessed. However, if this phase is "successful" — i.e., $Status[r]$ is not assigned the value 1 by any task — then at the end of the first phase, the proposed new value for each word is contained within the *val* field of that word.

The second phase of a MWCAS operation consists of only one statement: the CAS at line 18. This CAS attempts to both validate and commit the operation by resetting the value of $Status[r]$ from 0 to 2. By Definition 1, this CAS, if successful, atomically changes the current value of each accessed word to the desired new value. The third and final phase consists of lines 19 through 24. In this phase, each word w that is accessed by the MWCAS operation of T_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 T_r performs a subsequent MWCAS operation, reinitializing $Status[r]$ does not change the current value of any word.

Example. Figure 3 depicts the effects of a MWCAS operation m by task T_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. Note that the current value of each word is unchanged. Also, $Status[3]$ has been updated to indicate that task T_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 T_9 (which must be of higher-priority) with new value 56. $Status[4]$ is now 1, indicating the failure of T_4's operation. $Status[3]$ is still 1, indicating that T_3's operation has also failed. Observe that T_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.[5] □

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 as given in Definition 1. Note that the current value of each word accessed by the `MWCAS` procedure is computed within that procedure in the same way as in the `Read` procedure (see lines 4 through 9).

Although the above description conveys the basic idea of the implementation, there are some subtleties that we have not yet addressed. One such subtlety concerns the `Read` procedure. If this procedure is invoked by T_r to read word w, and if line 25 is executed when $w.pid = q \wedge w.count = c$ holds, then the value of $Val(w)$ potentially could be determined incorrectly if the values of $Status[q]$ or $Save[q, c]$ were to change during the execution of the `Read` procedure. However, $Status[q]$ and $Save[q, c]$ affect the value of $Val(w)$ only if $w.valid$ is false when line 25 is executed. As explained above, any `MWCAS` operation of task T_q that accesses word w "cleans up" in its third phase, thereby ensuring that $w.pid \neq q \vee w.valid$ holds upon termination of that operation. Thus, if $w.pid = q \wedge \neg w.valid$ holds when line 25 of the `Read` procedure is executed by task T_r, then it must be the case that T_r has preempted T_q as illustrated in Figure 4. Because T_r has been preempted, the value of $Save[q, c]$ cannot change during the execution of the `Read` procedure. Also, it must be the case that $Status[q] \neq 2$ holds during the execution of this procedure (the value of $Status[q]$ could potentially be changed by a higher-priority task from 0 to 1, but this does not affect the value of $Val(w)$).

[5]In these insets, and in subsequent similar figures, we only concern ourselves with statement executions that arise from invoking the procedures in Figure 2, i.e., we abstract away from the other statement executions of the tasks invoking these procedures. We denote operations by line segments, with time running from left to right.

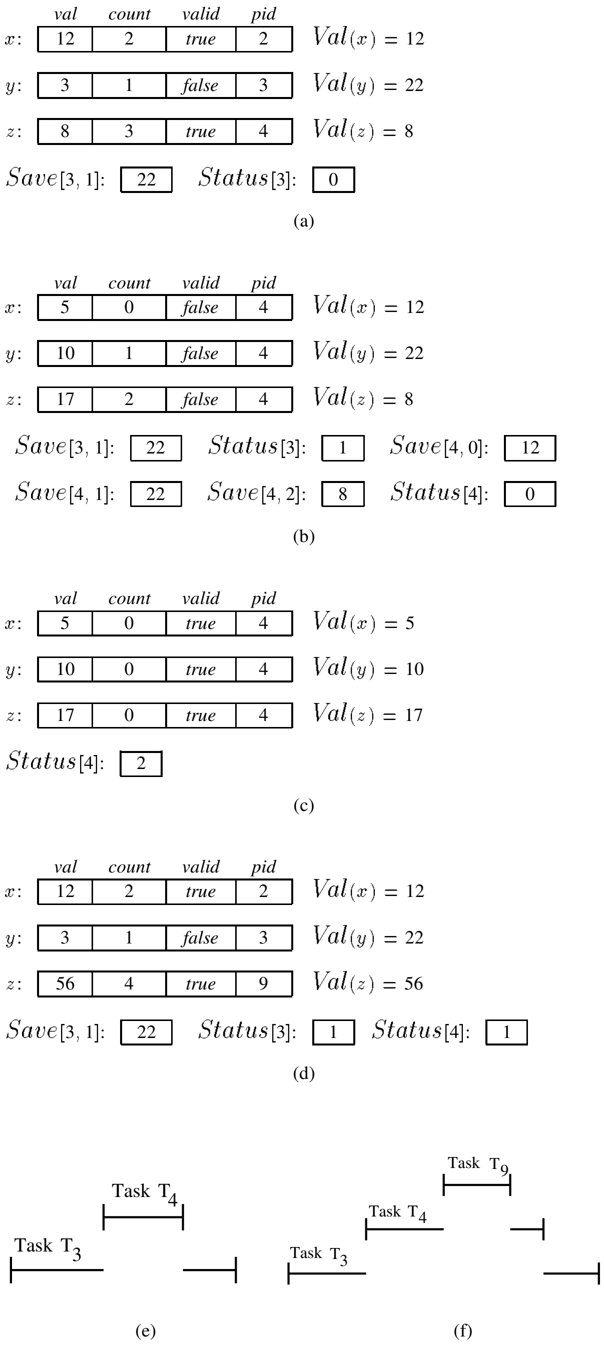

Figure 3. Task T_4 performs a `MWCAS` operation on words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. The contents of relevant shared variables are shown (a) at the beginning of the operation; (b) after the loop in lines 3..17; (c) at the end of the operation, assuming success; and (d) at the end of the operation, assuming failure on word z. The operation interleavings that result in (c) and (d) are shown in (e) (T_4 preempts T_3) and (f) (T_4 preempts T_3, and T_9 preempts T_4), respectively.

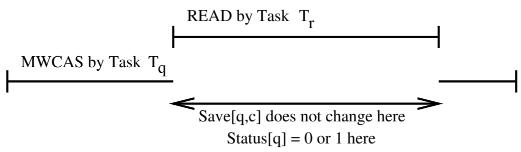

Figure 4. Example of a `Read` operation by Task T_r.

Another subtlety involves the conditions under which a `MWCAS` operation may fail. Strictly speaking, it should be possible to linearize any `MWCAS` operation to some point during its execution at which it "appears" to take effect [8]. Successful `MWCAS` operations can be linearized to the state at which line 18 is executed. To correctly linearize a failed `MWCAS` operation m, it is necessary that there be a state during the execution of m at which the current value of some word accessed by m differs from the old value specified for that word. In our implementation, we allow this linearization requirement to be violated by permitting a `MWCAS` m to fail if it is "overlapped" by a `MWCAS` operation m' that attempts to modify a word that is accessed by m. Observe that m' may itself fail, in which case it might not be possible to correctly linearize m as discussed above. In fact, this is exactly what happened in Figure 3(f). Task T_9 causes task T_4 to fail because T_9 modifies word z, which is accessed by T_4. Task T_3 is also caused to fail because it accesses word y, which T_4 attempts (unsuccessfully) to modify. Observe that it was not necessary (nor correct, strictly speaking) for T_3 to fail in this case. We allow a `MWCAS` operation to fail in a situation like this because it greatly simplifies the `MWCAS` algorithm and because the scheduling conditions derived in Section 3 view such a situation as an interference anyway.

A final subtlety involves `MWCAS` operations in which some word is accessed but not modified. Such an access should not interfere with accesses of that word by other tasks. To see how this is accomplished in our implementation, consider the situation in Figure 5(a). The only word in common between operations m_1 and m_2 is x. Neither operation changes the value of x, so both succeed. In particular, note that m_2 restores the value of x (line 22) before completing. Thus, it appears to m_1 that no other task has updated x. In contrast, consider the situation in Figure 5(b). In this situation, m_1 and m_2 are preempted by a third operation m_3 that does modify x. In this case, m_3 causes m_2 to fail by updating m_2's *Status* variable (line 13). m_2 in turn causes m_1 to fail by updating m_1's *Status* variable (line 23).

The full paper contains a detailed correctness proof of our `MWCAS`/`Read` implementation. From this proof and the code in Figure 2, we conclude the following.

Theorem 1: A `Read` operation and a W-word `MWCAS` operation can be implemented in a wait-free manner from `CAS` with $O(1)$ and $O(W)$ time complexity, respectively, in a real-time uniprocessor system. □

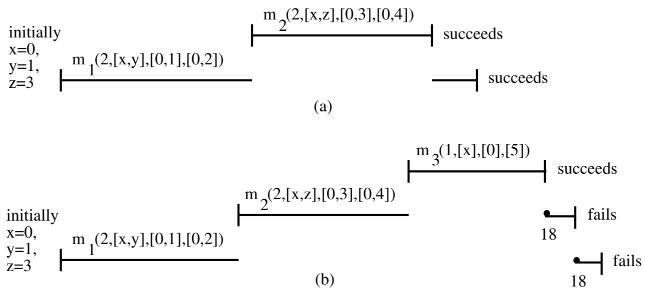

Figure 5. (a) Two overlapping `MWCAS` operations m_1 and m_2. Parameters are (*number of words*, [*list of words accessed*], [*list of old values*], [*list of new values*]). The only potential conflict is on word x, which neither m_1 nor m_2 changes, so both succeed. (b) m_1 and m_2 are overlapped by a third `MWCAS` operation, m_3, which does change x; m_1 and m_2 are preempted just before executing the `CAS` at line 18.

3. Scheduling with Lock-Free Objects

Scheduling conditions that apply to tasks that share lock-free objects can be obtained by modifying scheduling conditions for independent tasks to account for the overhead of operation interferences. In this section, we present a general approach based on linear programming for determining a bound on the cost of such interferences over an interval of time. We illustrate the utility of this approach by applying it to obtain scheduling conditions for the RM, DM, and EDF schemes, and a variation of the EDF scheme, which we call the EDF/NPD scheme, in which deadlines do not equal periods. Our approach for bounding the cost of interferences can be applied to any scheduling scheme satisfying the following axioms.

Axiom 1: If a job of task T_i can preempt a job of task T_j, then no job of T_j preempts any job of T_i. □

Axiom 2: The priority of a job does not change while accessing a shared object. □

Axiom 3: Different jobs of the same task cannot preempt one another. □

These axioms hold for the RM, DM, EDF, and EDF/NPD schemes, and for variations of these schemes in which tasks consist of multiple phases with separate execution priorities. We make explicit the preemption order between any two tasks — as specified by Axiom 1 — by indexing the tasks such that, if a job of task T_i can preempt a job of task T_j, then $i < j$. For the RM, DM, EDF, and EDF/NPD schemes, arranging tasks in the order of increasing deadlines results in task indices compatible with this indexing scheme.

3.1. Definitions and Notation

The scheduling conditions we derive are based on a preemptive, periodic task model in which all tasks are multiprogrammed on one processor. Most aspects of the model are in keeping with other similar models considered in the literature, and therefore are not described in detail here. One aspect that does warrant attention is the manner in which we model object accesses. We assume that each job is composed of distinct nonoverlapping computational fragments or *phases*. Each phase is either a *computation phase* or an *object-access phase*. Shared objects are not accessed during a computation phase. An object-access phase consists of exactly one retry loop in which one or more objects are accessed. The cost of an object-access phase is equal to the cost of its associated retry loop.

The following is a list of symbols that will be used repeatedly in deriving our scheduling conditions.

- N - The number of tasks in the system. We use i, j, and l as task indices; each is universally quantified over $\{0, \ldots, N-1\}$.
- p_i - The period of task T_i.
- $w(i)$ - The number of phases in a job of task T_i. The phases are numbered from 1 to $w(i)$. We use u and v to denote phases.
- c_i^v - The worst-case computational cost of the v^{th} phase of task T_i's job, where $1 \leq v \leq w(i)$, assuming no contention for the processor or shared objects. We denote total cost over all phases by $c_i = \sum_{v=1}^{w(i)} c_i^v$.
- $m_j^{i,v}(t)$ - The worst-case number of interferences in T_i's v^{th} phase due to T_j in an interval of length t.
- f_i^v - An upper bound on the number of interferences of the retry loop in the v^{th} phase of T_i during a single execution of that phase.

We obtain scheduling conditions by determining the worst-case demand of each task. The *demand due to task* T_i in an interval is the total computation time required by jobs of T_i in that interval. In our analysis, we assume that an object-access phase is interfered with every time a higher-priority job that modifies a common object is released during that phase. This assumption is pessimistic because not all such releases necessarily cause an interference. If a job of T_j interferes with the v^{th} phase of a job of T_i, then an additional demand is placed on the processor, because another execution of the retry-loop iteration in T_i's v^{th} phase is required. We denote this additional demand by $s_j^{i,v}$. Formally, $s_j^{i,v}$ is defined as follows.

Definition 2: Let T_i and T_j be two distinct tasks, where T_i has at least v phases. Let z_j denote the set of objects modified by T_j, and a_i^v denote the set of objects accessed in the v^{th} phase of T_i. Then,

$$s_j^{i,v} = \begin{cases} c_i^v & \text{if } j < i \wedge a_i^v \cap z_j \neq \emptyset \\ 0 & \text{otherwise.} \end{cases} \qquad \square$$

3.2. Bounding Interference Cost

The goal of this subsection is to show how to obtain a reasonably accurate bound on the additional demand due to interferences over an interval $\mathcal{I}$. To this end, we define an expression that gives the *exact* worst-case cost of interferences in tasks T_0 through T_i in any interval of length t.

Definition 3: The total cost of interferences in jobs of tasks T_0 through T_i in any interval of length t, denoted $E_i(t)$, is defined as follows: $E_i(t) \equiv \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) s_l^{j,v}$. $\square$

The term $m_l^{j,v}(t)$ in the above expression denotes the worst-case number of interferences caused in T_j's v^{th} phase by jobs of T_l in an interval of length t. The term $s_l^{j,v}$ represents the amount of additional demand required if T_l interferes once with T_j's v^{th} phase. The expression within the leftmost summation denotes the total cost of interferences in a task T_j over all phases of all jobs of T_j in an interval of length t.

Expression $E_i(t)$ accurately reflects the worst-case additional demand placed on the processor in an interval $\mathcal{I}$ of length t due to interferences in tasks T_0 through T_i. Of course, to evaluate this expression, we first must determine values for the $m_l^{j,v}(t)$ terms. Unfortunately, in order to do so, we potentially have to examine an exponential number of possible task interleavings in the interval $\mathcal{I}$. Instead of exactly computing $E_i(t)$, our approach is to obtain a bound on $E_i(t)$ that is as tight as possible. We do this by viewing $E_i(t)$ as an expression to be maximized. The $m_l^{j,v}(t)$ terms are the "variables" in this expression. These variables are subject to certain constraints. We obtain a bound for $E_i(t)$ by using linear programming to determine a maximum value of $E_i(t)$ subject to these constraints. We now explain how appropriate constraints on the $m_l^{j,v}(t)$ variables are obtained. In this explanation, we focus on the RM scheme. At the end of this subsection, we explain how similar constraints can be obtained for other schemes.

We impose three sets of constraints on the $m_j^{i,v}(t)$ variables. All of these constraints are straightforward. However, the third constraint involves terms (f_i^v) that are not completely straightforward to calculate. Most of the rest of this subsection is devoted to explaining how these terms are computed. For a set of tasks scheduled under the RM scheme, and an interval of length t, the three sets of constraints are as follows.

Constraint Set 1:
$(\forall i, j : j < i :: \sum_{v=1}^{w(i)} m_j^{i,v}(t) \leq \left\lceil \frac{t+1}{p_j} \right\rceil)$. □

Constraint Set 2:
$(\forall i :: \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) \leq \sum_{j=0}^{i-1} \left\lceil \frac{t+1}{p_j} \right\rceil)$. □

Constraint Set 3:
$(\forall i, v :: \sum_{j=0}^{i-1} m_j^{i,v}(t) \leq \left\lceil \frac{t+1}{p_i} \right\rceil f_i^v)$. □

The first set of constraints follows because the number of interferences in jobs of T_i due to T_j in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of T_j that can be released in $\mathcal{I}$. The second set of constraints follows from a result presented in [3], which states that the total number of interferences in all jobs of tasks T_0 through T_i in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of tasks T_0 through T_{i-1} released in $\mathcal{I}$. In the third set of constraints, the term f_i^v is an upper bound on the number of interferences of the retry loop in the v^{th} phase of T_i during a single execution of that phase. The details of calculating f_i^v are described later. The reasoning behind this set of constraints is as follows. If at most f_i^v interferences can occur in the v^{th} phase of a job of T_i, and if there are n jobs of T_i released in an interval $\mathcal{I}$, then at most nf_i^v interferences can occur in the v^{th} phase of T_i in $\mathcal{I}$.

We use an inductive approach to calculate f_i^v for any i and v. This inductive approach is expressed in pseudo-code in Figure 6. The *compute_retries* procedure in this figure computes all f_i^v values. This procedure begins by setting f_0^v to zero for all phases of T_0 (line 1). This is because, by Axiom 1 and our ordering on tasks, operations of T_0 can never be interfered with. We then calculate the f values for tasks T_1 through T_{N-1}, respectively. If the v^{th} phase of task T_i is a computation phase, then f_i^v is set to zero (line 5) because a computation phase cannot be interfered with. Lines 6 through 10 are executed if the v^{th} phase of task T_i is an object-access phase. In this case, we first calculate a bound R_1 on the maximum time it takes to execute phase v, given that at most k interferences of phase v can occur (line 7). We then calculate a bound R_2 on the maximum time it takes to execute phase v, given that at most $k+1$ interferences of phase v can occur (line 8). The manner in which R_1 and R_2 are determined is described below. If R_2 exceeds the period of task T_i, then we have failed to find a constraint that can be imposed on the number of interferences in phase v (line 9). If R_1 equals R_2, then phase v of T_i can experience at most k interferences (line 10).

We now explain the manner in which R_1 is determined; R_2 is calculated in a similar manner. R_1 is assigned a value t that is an upper bound on the length of an interval that includes $n \leq k+1$ iterations of phase v of task T_i; the interval begins with the first statement execution in the first iteration of phase v, and ends with the last statement execution of the n^{th} execution of phase v. In line 7, the first component in the left-hand side of the inequality denotes the portion of time in the interval that is taken to execute the last iteration of phase v. The second component denotes the time spent executing jobs of higher-priority tasks, excluding interferences in those tasks. (In the interval of length t in question, there can be no higher-priority job releases at the first point in the interval, and any such job released at the $(t+1)^{st}$ point in the interval executes after the interval. This is why $t-1$ appears in this expression.) The third component denotes an upper bound on the additional time spent executing additional iterations of loops that have been interfered with in T_i's v^{th} phase and in higher-priority tasks.

The third component is calculated by invoking *ic*(i, v, k, t), which determines an upper bound on the interference cost in tasks T_0 through T_{i-1} and the v^{th} phase of task T_i in an interval of length t in which T_i is interfered with at most k times. Determining an exact bound is difficult, so we use linear programming within *ic* to obtain an upper bound. The constraints for *ic*(i, v, k, t) only use f values of tasks T_0 through T_{i-1}, so there is no circularity.

The maximum value of the expression given in *ic* is determined subject to five constraint sets, labeled (a) through (e). The set of constraints labeled (a) follows from our definition of the interval to be determined. For example, for R_1, the interval ends with the completion of the n^{th} iteration of phase v of task T_i, where $n \leq k+1$. The set of constraints labeled (b) follows from the fact that the number of times a higher-priority task T_j can interfere with T_i's v^{th} phase in an interval is bounded by the number of jobs of T_j released in that interval. The rest of the constraint sets are similar to Constraint Sets 1 through 3 given earlier. In the full paper, we show that the bounds returned by *compute_retries* are correct by proving the following lemma.

Lemma 1: The value returned for f_i^v by *compute_retries* is an upper bound on the number of times the v^{th} phase of T_i can be interfered with in a single job of T_i. □

The constraints considered so far apply not only to RM scheduling, but also to DM scheduling. Similar constraints can be derived for the EDF and EDF/NPD schemes. In particular, Constraint Sets 1 through 3 can be transformed to EDF constraint sets by replacing the ceiling function in the bounding terms by a floor function, and by replacing $t+1$ in the numerator of the bounding terms by t. For example, Constraint Set 2 is transformed to the EDF constraint set $(\forall i :: \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) \leq \sum_{j=0}^{i-1} \lfloor t/p_j \rfloor)$. This follows because the number of jobs released in an interval of length t with deadlines at or before t equals $\lfloor t/p_j \rfloor$. Constraints for EDF/NPD scheduling can be derived from those for EDF scheduling by accounting for the deadline d_i (where $d_i \leq p_i$) of each task T_i, e.g., the previous EDF con-

procedure *compute_retries*()

for v := 1 **to** $w(0)$ **do** f_0^v := 0 **od**; $/*$ Task T_0 cannot be interferred with $*/$
for i := 1 **to** $N-1$ **do**
  for v := 1 **to** $w(i)$ **do** $/*$ Consider each phase v of task T_i $*/$
    if T_i's v^{th} phase is a computational phase **then**
      $f_i^v := 0$ $/*$ Computational phase cannot be interferred with $*/$
    else $/*$ Phase v is an object-access phase $*/$
      for $k := 0$ **to** ∞ **do**
        $R_1 := (min\ t :: c_i^v + \sum_{j=0}^{i-1} \lceil (t-1)/p_j \rceil c_j + ic(i,v,k,t-1) \leq t)$;
        $R_2 := (min\ t :: c_i^v + \sum_{j=0}^{i-1} \lceil (t-1)/p_j \rceil c_j + ic(i,v,k+1,t-1) \leq t)$;
        if $R_2 \geq p_i$ **then** $f_i^v := \infty$; **break fi**; $/*$ Period exceeded, give up $*/$
        if $R_2 = R_1$ **then** $f_i^v := k$; **break fi** $/*$ At most k interferences can occur $*/$
      od
    fi
  od
od

procedure $ic(i,v,k,t)$ **returns integer** $/*$ Computes bound on **i**nterference **c**osts in T_i and higher-priority tasks in an interval of length t during the v^{th} phase of T_i in which T_i is interfered with at most k times $*/$

return the maximum value of

$\sum_{j=0}^{i-1} m_j^{i,v}(t) s_j^{i,v} + \sum_{j=0}^{i-1} \sum_{u=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,u}(t) s_l^{j,u}$

subject to the following constraints:

(a) $\sum_{j=0}^{i-1} m_j^{i,v}(t) \leq k$

(b) $(\forall j : 0 \leq j < i :: m_j^{i,v}(t) \leq \left\lceil \frac{t+1}{p_j} \right\rceil)$

(c) $(\forall j, l : j < l < i :: \sum_{u=1}^{w(l)} m_j^{l,u}(t) \leq \left\lceil \frac{t+1}{p_j} \right\rceil)$

(d) $(\forall l : l < i :: \sum_{j=0}^{l} \sum_{v=1}^{w(j)} \sum_{l'=0}^{j-1} m_{l'}^{j,v}(t) \leq \sum_{j=0}^{l-1} \left\lceil \frac{t+1}{p_j} \right\rceil)$

(e) $(\forall l, u : 0 \leq l < i :: \sum_{j=0}^{l-1} m_j^{l,u}(t) \leq \left\lceil \frac{t+1}{p_l} \right\rceil f_l^u)$

Figure 6. Pseudo-code to calculate f_i^v values.

straint set becomes $(\forall i :: \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) \leq \sum_{j=0}^{i-1} \lfloor (t + p_i - d_i)/p_j \rfloor)$. For the EDF and EDF/NPD schemes, the *compute_retries* and *ic* procedures are the same as for the RM scheme. (Ceilings must be used instead of floors in these procedures because a task's deadline is not considered when bounding interferences in its phases.)

3.3. Scheduling Conditions

We now present sufficient scheduling conditions for the RM, DM, EDF, and EDF/NPD schemes. In each of these conditions, we give an expression that represents the maximum demand in an interval $\mathcal{I}$ of length t, and require that total demand over $\mathcal{I}$ is less than or equal to the available processor time in $\mathcal{I}$. The expression for demand consists of two components: the first represents demand due to job releases, and the second represents demand due to interferences. Recall that $E_i(t)$ is the actual worst-case cost of interferences in jobs of tasks T_0 through T_i in any interval of length t. We let $E_i'(t)$ denote a bound on $E_i(t)$ that is determined as described in the previous subsection. The scheduling condition for the RM scheme is as follows.

Theorem 2: Under the RM scheme, a set of tasks is schedulable if the following holds for every task T_i.

$$(\exists t : 0 < t \leq p_i :: \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + E_i'(t-1) \leq t)$$

Proof: We prove that if a task set is not schedulable, then the negation of the above expression holds. Let the k^{th} job of some task T_i be the first to miss its deadline, and let $r_i(k)$ denote the time at which this job is released, i.e., $r_i(k) = r_i(0) + kp_i$. The proof hinges on examining the *busy point* of the k^{th} job of task T_i, which is denoted by $b_i(k)$. The busy point $b_i(k)$ is the most recent point in time at or before $r_i(k)$ when T_i and all higher-priority jobs either release a job or have no unfulfilled demand. In [3], it is shown that "if the k^{th} job of T_i misses its deadline, and if t is some point in $[b_i(k), r_i(k+1))$, then the difference between the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$ and the available processor time in that interval is greater than one". Thus, for any t in $[b_i(k), r_i(k+1))$, we have the following.

$$\sum_{j=0}^{i} \left\lceil \frac{t - b_i(k) + 1}{p_j} \right\rceil c_j + E_i(t - b_i(k)) > t - b_i(k) + 1 \quad (1)$$

Observe that (1) is independent of t and $b_i(k)$ (it is a function of the length of an interval). Thus, we can replace $t - b_i(k)$ in (1) by t', where $t' = t - b_i(k)$ and $t' \in [0, r_i(k+1) - b_i(k))$ to get $\sum_{j=0}^{i} \left\lceil \frac{t'+1}{p_j} \right\rceil c_j + E_i(t') > t' + 1$. Now, replace t' by t, where $t = t' + 1$ and $t \in (0, r_i(k+1) - b_i(k)]$. Then, we have $\sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + E_i(t-1) > t$ for all t in $(0, r_i(k+1) - b_i(k)]$. Finally, note that $E_i'(t-1) \geq E_i(t-1)$. Hence, we have the following.

$$\sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + E_i'(t-1) > t \quad (2)$$

By definition, $b_i(k) \leq r_i(k)$. Therefore, the interval $(0, r_i(k+1) - r_i(k)]$ is completely contained in $(0, r_i(k+1) - b_i(k)]$. Because $r_i(k+1) - r_i(k) = p_i$, (2) therefore holds for all t in $(0, p_i]$. □

Sufficient scheduling conditions can be proved for the DM, EDF, and EDF/NPD schemes in a manner similar to that above. We state these conditions without proof in the following theorems.

Theorem 3: Under the DM scheme, a set of tasks is schedulable if the following holds for every task T_i.

$$(\exists t : 0 < t \leq d_i :: \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + E_i'(t-1) \leq t) \quad □$$

Theorem 4: Under the EDF scheme, a set of tasks is schedulable if the following holds.

$(\forall t :: \sum_{j=0}^{N-1} \left\lfloor \frac{t}{p_j} \right\rfloor c_j + E'_{N-1}(t-1) \leq t)$ □

Theorem 5: Under the EDF/NPD scheme, a set of tasks is schedulable if the following holds.

$(\forall t :: \sum_{j=0}^{N-1} \left\lfloor \frac{t+p_j-d_j}{p_j} \right\rfloor c_j + E'_{N-1}(t-1) \leq t)$ □

As formulated above, the expressions in Theorems 4 and 5 cannot be verified because the value of t is unbounded. However, there is an implicit bound on t. In particular, we only need to consider values less than or equal to the least common multiple (LCM) of the task periods. (If an upper bound on the utilization available for the tasks is known, then we can restrict t to a much smaller range [6].)

4. Simulation Results

In this section, we present results from simulation experiments conducted to compare lock-free, wait-free, and lock-based object implementations under the RM scheme. These experiments involved randomly generated task sets consisting of 10 tasks and 5 shared objects, obtained by varying four parameters: r/w ratio, cost ratio, conflicts, and nesting level. The *r/w ratio* parameter specifies the fraction of all operations that are read-only. This parameter is of interest because, as explained in Section 2.1, read-only operations do not interfere with each other in lock-free implementations. The *cost ratio* parameter specifies the ratio of the cost of a lock-free (wait-free) object access to that of a lock-based access; e.g., the retry-loop cost of a lock-free object is (on average) twice as expensive as a lock-based access if the cost ratio parameter is 2. The cost of a lock-based operation includes the cost of acquiring and releasing a lock; for lock-based objects, an implementation based on the stack resource policy was assumed [4]. If the *conflicts* parameter is k, then at least one object is accessed by k tasks, and no object is accessed by more than k tasks. In our experiments, tasks were modeled as a sequence of three phases, of which only the second is an object-access phase. The *nesting level* parameter specifies the number of objects accessed in this phase, and ranges from 1 to 3 (the word "nesting" refers to nested locks in lock-based implementations).

In order to bound the simulation lengths, task periods were randomly selected from a predetermined set of 36 periods. For the set of periods considered, the LCM of the periods was 134,534,400 time units, and the minimum and maximum periods were 8,448 and 1,747,200 time units, respectively. Computation phase costs ranged between 1 and 500 time units, and were randomly generated subject to the constraint that overall utilization is at most one. In all experiments, context switch times were ignored.

Lock-based object access costs were randomly generated assuming a normal distribution with mean and standard deviation of 128 and 20 time units, respectively. The overall cost of each object-access phase depends on both the nesting level and the object access costs. In our experiments, nesting levels 1, 2, and 3 were selected with probability 0.6, 0.25, and 0.15, respectively. We selected this distribution based on our belief that multi-object accesses are less frequent than single-object accesses in practice.

To compare the performance of the different schemes, we calculated the breakdown (computation) utilization of each task set that was generated. The *breakdown utilization* (BU) of a task set is obtained by scaling the cost of task phases, and is defined to be the maximum utilization at which the task set is still schedulable. The total utilization of all computation phases of all tasks at the breakdown point is called the *breakdown computation utilization* (BCU).

BU and BCU curves resulting from our experiments are shown in Figures 7 and 8. Each curve in these figures was obtained from 4,000 generated task sets. Experimental BU (BCU) values for lock-based and lock-free schemes are given by "blocking_experimental" and "lockfree_experimental", respectively. These values were obtained for each generated task set by checking schedulability in a brute force manner, i.e., by checking to the LCM of the task periods. Predicted BU (BCU) values for lock-based objects are given by "blocking_predicted". Values for this case were obtained by using the scheduling condition given in [15]. BU (BCU) values predicted by the scheduling conditions presented in this paper and in [3] are given by "lockfree_predicted_new" and "lockfree_predicted_old", respectively. Observe that the RM scheduling condition presented in this paper is much tighter than that given in [3]. Also, the new condition results in better predications when there are fewer conflicts, and (although not shown here) when most operations are read-only. BU (BCU) values for wait-free objects are given by "waitfree"; these values were obtained by using the RM scheduling condition in [11]. Experimental BCU values are not tabulated for this case because the RM condition in [11] is necessary and sufficient.

Our simulations indicate that only the cost ratio parameter significantly affects relative performance. Simulations for different r/w ratios and other nesting probabilities (not shown here) resulted in similar graphs. In examining the effects of various cost ratios, it is best to focus on BCU values. This is because the BU curves include overhead associated with object accesses, and because the experimental BU curves do not show much variation. (A high BU curve can be misleading, because much of the utilization accounted for in the BU values may be due to object sharing overhead; an inefficient object sharing scheme may give rise to high BU values solely because of this overhead.) The BCU curves in Figure 8 indicate that lock-free objects perform better than lock-based schemes when the cost ratio is less than one (inset (a)), slightly worse than lock-based schemes when the cost ratio equals one (inset (b)), and worse than lock-based

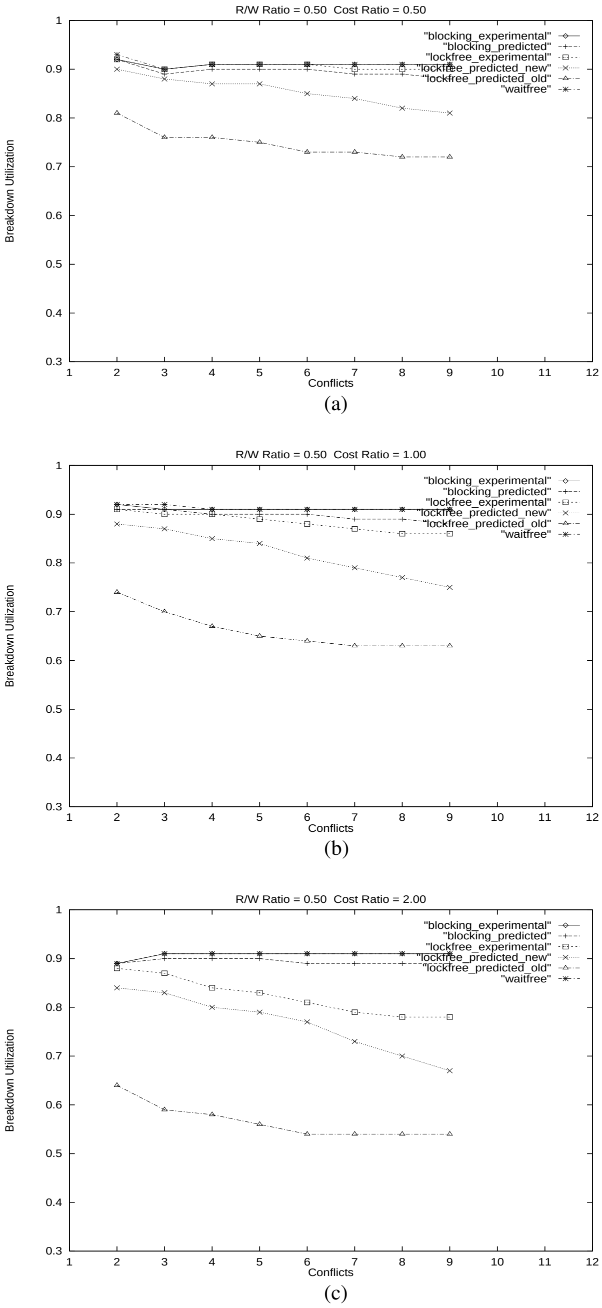

Figure 7. Breakdown Utilization.

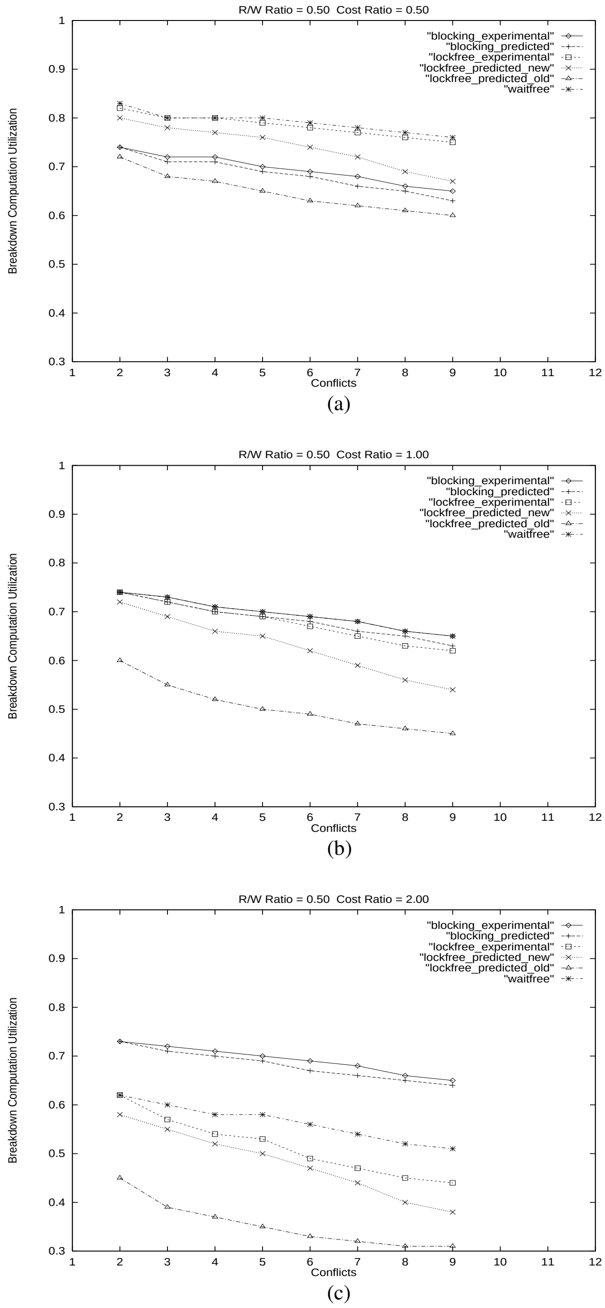

Figure 8. Breakdown Computation Utilization.

schemes when the cost ratio is greater than one (inset (c)).

The main conclusion to be drawn from these experiments is that, when lock-free loop costs are (on average) less than corresponding lock-based access costs, lock-free implementations perform better. Preliminary cost figures from actual implementations indicate that lock-free implementations of common objects 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 objects like balanced trees are likely to be more efficient than lock-free ones. Wait-free implementations perform better than their lock-free counterparts in all situations 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 algorithmic overhead required to ensure wait-freedom. On the other hand, for certain very simple objects like read/write buffers, a wait-free implementation may be the best choice.

Although our results indicate that the r/w ratio parameter is not very significant, in practice, a high r/w ratio will result in a low cost ratio for lock-free objects. This is because, for many objects, read-only operations do not require copying.

Thus, lock-free implementations may be preferable if most operations are read-only. In our experiments, we did not account for the fact that read-only operations are usually more efficient than updates in lock-free implementations. In addition, the larger conflicts parameter values considered (which resulted in less accurate predications for lock-free) may not be reflective of what one would find in practice.

5. Concluding Remarks

We have presented an integrated framework for the development of real-time applications in which tasks share lock-free objects. This framework consists of two key components. The first is an efficient implementation of a `MWCAS` primitive for real-time uniprocessor applications. This primitive can be used to simplify the implementation of many lock-free objects, and to implement multi-object operations and transactions. The second key component of our framework is a general approach, based on linear programming, for determining bounds on the cost of operation interferences when lock-free objects are used. The simulation studies we have presented suggest that this approach is likely to produce reasonably accurate bounds in practice.

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*, ACM, New York, August 1995, pp. 184-193.

[2] 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.

[3] J. Anderson, S. Ramamurthy and K. Jeffay, "Real-Time Computing with Lock-Free Shared Objects (Extended Abstract)", *Proceedings of the 16th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, December 1995, pp. 28-37.

[4] T. Baker, "Stack-Based Scheduling of Real-Time Processes", *Journal of Real-Time Systems*, 3(1), March 1991, pp. 67-99.

[5] 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.

[6] S. Baruah, R. Howell, and L. Rosier, "Feasibility Problems for Recurring Tasks on One Processor", *Theoretical Computer Science*, 118, 1993, pp. 3-20.

[7] M. I. Chen and K. J. Lin, "Dynamic Priority Ceiling: A Concurrency Control Protocol for Real Time Systems", *Journal of Real-Time Systems*, 2(1), 1990, pp. 325-346.

[8] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990, pp. 463-492.

[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*, ACM, New York, August 1994, pp. 151-160.

[10] K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard Real-Time Systems", *Proceedings of the 13th IEEE Symposium on Real-Time Systems*, Phoenix, AZ, 1992, pp. 89-99.

[11] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, Santa Monica, CA, 1989, pp. 166-171.

[12] J.Y.T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks", *Performance Evaluation*, 2(4), 1982, pp. 237-250.

[13] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, 30, Jan. 1973, pp. 46-61.

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

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

[16] 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*, ACM, New York, May 1996, pp. 233-242.

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

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