Real-Time Computing with Lock-Free Shared Objects

JAMES H. ANDERSON, SRIKANTH RAMAMURTHY, and KEVIN JEFFAY
University of North Carolina

This paper considers the use of lock-free shared objects within hard real-time systems. As the name suggests, *lock-free* shared objects are distinguished by the fact that they are accessed without locking. As such, they do not give rise to priority inversions, a key advantage over conventional, lock-based object-sharing approaches. Despite this advantage, 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 object implementations permit concurrent operations to interfere with each other, and repeated interferences can cause a given operation to take an arbitrarily long time to complete.

The main contribution of this paper is to show that such interferences can be bounded by judicious scheduling. This work pertains to periodic, hard real-time tasks that share lock-free objects on a uniprocessor. In the first part of the paper, scheduling conditions are derived for such tasks, for both static and dynamic priority schemes. Based on these conditions, it is formally shown that lock-free shared objects often incur less overhead than object implementations based on wait-free algorithms or lock-based schemes. In the last part of the paper, this conclusion is validated experimentally through work involving a real-time desktop videoconferencing system.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, mutual exclusion, scheduling, synchronization*; J.7 [**Computer Applications**]: Computers in Other Systems—*real-time*

General Terms: Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: critical sections, deadline-monotonic, earliest-deadline-first, hard real-time, lock-free, rate-monotonic, scheduling, synchronization, wait-free

A preliminary version of this paper appeared in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, December 1995, pp. 28-37. The first author was supported by NSF grants CCR 9216421 and CCR 9510156, by an Alfred P. Sloan Research Fellowship, and by a U.S. Army Research Office Young Investigator Award, grant number DAAH04-95-1-0323. The second author was also supported by ARO grant DAAH04-95-1-0323. The third author was supported by NSF grant CCR 9510156 and by grants from the Intel and IBM corporations.
Address: Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175; email: {anderson, ramamurt, jeffay}@cs.unc.edu.

1. INTRODUCTION

Most work on implementing shared objects in preemptive hard real-time uniprocessor systems has focused on using critical sections to ensure object consistency. One of the main problems 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. In a hard real-time system, unless priority inversions are carefully controlled, it may be difficult or impossible to ensure that task deadlines are always met. Most conventional mechanisms for controlling priority inversions rely on the kernel to dynamically adjust task priorities to ensure that a task within a critical section executes at a priority that is sufficiently high to bound the duration of any priority inversion. Schemes that work in this way include priority inheritance protocol [Sha et al. 1990; Rajkumar 1991], the priority ceiling protocol (PCP) [Sha et al. 1990; Baker 1991; Rajkumar 1991], the dynamic PCP [Chen and Lin 1990], and the earliest-deadline-first scheme with dynamic deadline modification (EDF/DDM) [Jeffay 1992]. Although these schemes provide a general framework for real-time synchronization, this generality comes at a price, specifically added operating system overhead.

In this paper, we consider interprocess communication in object-based, hard real-time systems. Our main contribution is to show that *lock-free* shared objects [Lamport 1977; Massalin 1992; Bershad 1993; Herlihy 1993] — i.e., objects that are not critical-section-based — are a viable alternative to lock-based schemes such as the PCP in such systems. We establish this through a combination of formal analysis and experimentation. We begin by establishing scheduling conditions for hard real-time, periodic tasks that share lock-free objects on a uniprocessor under either rate-monotonic (RM) or earliest-deadline-first (EDF) scheduling [Liu and Layland 1973]. We also briefly present a slight variation of our RM condition that holds for the deadline-monotonic (DM) priority scheme [Leung and Whitehead 1982], and an EDF condition that holds when deadlines and periods do not coincide. (See Table 1 for a description of the scheduling schemes considered in this paper.) After presenting our scheduling conditions, we compare lock-free and lock-based approaches for implementing objects, both formally, based on our scheduling conditions, and experimentally, based on work involving a real-time desktop videoconferencing system. We also compare lock-free objects with wait-free objects. As explained below, wait-free objects are also implemented without critical sections, but are required to satisfy some additional conditions not required of lock-free objects.

Our formal analysis and experimental work both lead to the same conclusion: lock-free shared objects often incur less overhead than object implementations based on wait-free algorithms or lock-based schemes. In addition, our scheduling conditions show that lock-free objects can be applied without detailed knowledge of which specific tasks access which objects. This makes them easier to apply than lock-based schemes. Also, with lock-free objects, new tasks can be added dynamically to a system with very little effort. In contrast, adding new tasks with lock-based schemes entails recomputing certain operating system tables (e.g., tables required by the PCP to record the highest-priority task that locks each semaphore).

Lock-free operations are usually implemented using “retry loops”. Figure 1 depicts a lock-free enqueue operation that is implemented in this way. An item is

Table 1. Real-Time Scheduling Schemes

Scheme	Definition[a]
Deadline-Monotonic (DM)	A static-priority scheme in which tasks with shorter relative deadlines[b] have higher priorities.
Rate-Monotonic[c] (RM)	A static-priority scheme in which tasks with shorter periods have higher priorities.
Earliest-Deadline-First[c] (EDF)	A dynamic-priority scheme in which, at any given instant, the task that has the closest deadline has highest priority.

[a] These schemes are *priority-based*, i.e., the processor always executes the highest-priority task available for execution. A *periodic* task is a sequential program that is invoked repeatedly. Successive invocations are separated by a constant amount of time, called the task's *period*.

[b] The *relative deadline* of a task is the elapsed time between the beginning of an invocation of that task and the deadline of that invocation. A task's relative deadline is assumed to be less than or equal to its period.

[c] Under the RM and EDF schemes, it is assumed that task deadlines and periods coincide, i.e., each invocation of a task must complete execution before the next invocation of that task begins. (At the end of Section 4, however, we do consider EDF scheduling when deadlines and periods do not coincide.)

enqueued in this implementation by using a two-word compare-and-swap (CAS2) instruction[1] to atomically update a tail pointer and either the "next" pointer of the last item in the queue or a head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the CAS2 instruction succeeds. Note that the queue is not actually "locked" by any task. An important property of lock-free implementations such as this is that operations may *interfere* with each other. An interference results in the enqueue example when a successful CAS2 by one task results in a failed CAS2 by another task.

In this paper, we use the term "lock-free" to refer to object implementations based on an unbounded retry loop structure like that depicted in Figure 1.[2] Some lock-free implementations do not adhere to this characterization. For example, there exists an important special class of lock-free implementations known as *wait-free* implementations [Peterson 1983; Lamport 1986; Herlihy 1991; Herlihy 1993] in which operations must satisfy a strong form of lock-freedom that precludes all waiting dependencies among tasks, including potentially unbounded retry loops.[3] Although one motivation for work on wait-free objects has been their potential use in real-time systems, our results show that lock-free objects are often superior for real-time computing on uniprocessors.

From a real-time perspective, lock-free object implementations are of interest because they avoid priority inversion and deadlock with no underlying operating

[1] The first two parameters of CAS2 specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed. Note that it is possible to simulate the CAS2 instruction in software, as discussed in Section 7.
[2] Some authors use the term "nonblocking" to refer to such implementations.
[3] More precisely, individual wait-free operations are required to be starvation-free. In contrast, lock-free objects guarantee only system-wide progress: if several tasks concurrently access such an object, then *some* access will eventually complete.

type *Qtype* = **record** *data*: *valtype*; *next*: **pointer to** *Qtype* **end**
shared variable *Head*, *Tail*: **pointer to** *Qtype*

procedure *Enqueue*(*input*: *valtype*)
private variable *old*, *new*: **pointer to** *Qtype*;
addr: **pointer to pointer to** *Qtype*
begin
$*new$:= (*input*, *NULL*);
repeat *old* := *Tail*;
if $old \neq NULL$ **then** *addr* := &(*old*->*next*) **else** *addr* := &*Head* **fi**
until CAS2(&*Tail*, *addr*, *old*, *NULL*, *new*, *new*)
end

Fig. 1. Lock-free enqueue implementation.

system support for object sharing. 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. Nonetheless, we show that if tasks on a uniprocessor are scheduled appropriately, then lock-free retry loops are indeed bounded. 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 T_i and T_j that access a common lock-free object B. Suppose that T_i causes T_j to experience a failed update of B. On a uniprocessor, this can only happen if T_i preempts the access of T_j and then updates B successfully. Thus, there is a correlation between failed updates and task preemptions. The maximum number of task preemptions within a time interval can be determined from the timing requirements of the tasks. Using this information, it is possible to determine a bound on the number of failed updates in that interval. Intuitively, a set of tasks that share lock-free objects is schedulable if there is enough free processor time to accommodate the failed updates that can occur over any interval.

The formal analysis that we present establishes a fundamental tradeoff between lock-free and lock-based approaches. This tradeoff essentially hinges on the cost of a lock-free retry loop, and the cost of the operating system overhead that arises in lock-based schemes (which can be substantial). An important question, then, is how costly lock-free retry loops are likely to be. Any general methodology for constructing lock-free objects must be based on "universal" lock-free constructions [Herlihy 1991; Herlihy 1993]. Such a construction can be employed to implement *any* object in a lock-free manner. Unfortunately, this generality can lead to expensive implementations. For example, in the universal lock-free construction of [Herlihy 1993], numerous copies of the implemented object are kept. The "current" copy is indicated by a shared object pointer. Each task's retry loop consists of the following steps: first, the shared object pointer is read using a *load-linked* operation and a local copy of the object is made; then, the desired operation is performed on the local copy; finally, a *store-conditional* operation is performed to attempt to make the shared object pointer to point to the local copy. This retry loop can be expensive for certain large objects due to the overhead of copying. Fortunately, techniques have been recently developed that can be applied to substantially reduce this copying overhead [Alemany and Felten 1992; Anderson and Moir 1995]. Also, as shown in [Massalin 1992], many common objects, including most that would be of use in a real-time system, can be implemented with very short retry loops, such as that depicted in Figure 1. Finally, recent work by the first two authors and colleagues has shown that, in real-time systems, the priority structure that exists often can be exploited to simplify object implementations, eliminating copying overhead entirely [Ramamurthy et al. 1996; Anderson et al. 1997].

The lock-free approach to real-time object sharing that we espouse is actually rooted in work done by Sorenson and Hamacher in the real-time systems community some twenty years ago [Sorensen 1974; Sorensen and Hemachar 1975]. Sorenson and Hamacher's work involved a real-time communication mechanism based on wait-free read/write buffers. In their approach, all buffer management is done within the operating system, so it suffers from many of the same shortcomings as conventional lock-based approaches.

Unfortunately, the thread of research on wait-free and lock-free communication begun by Sorenson and Hamacher was lost in the real-time systems community for many years. Recently, however, this thread of research resurfaced in [Kopetz and Reisinger 1993] and in [Johnson and Harathi 1994]. In the former paper, a simple lock-free, one-writer, read/write buffer is presented, and scheduling conditions are given for tasks sharing the buffer. In the latter paper, the primary focus is implementations of lock-free algorithms rather than scheduling. Our work deals almost exclusively with scheduling, and significantly extends the work of Kopetz and Reisinger by focusing on arbitrary task sets and objects.

The rest of this paper is organized as follows. In Section 2, we present definitions and notation, and prove some key lemmas. We use these lemmas to derive scheduling conditions for the RM and DM priority schemes in Section 3, and the EDF priority scheme in Section 4. We then compare the overhead of lock-free synchronization with that of several other approaches, on a formal basis in Section 5, and on an experimental basis in Section 6. In these comparisons, we consider lock-based objects implemented using the PCP [Rajkumar 1991] and the EDF/DDM scheme [Jeffay 1992], and wait-free objects implemented using the constructions of [Herlihy 1993]. We end with concluding remarks in Section 7.

2. PRELIMINARIES

We use the term *task* to refer to a sequential program that is invoked repeatedly. A single execution of a task is called a *job*. The time at which a job arrives for execution is called its *release time*. A task is *periodic* if and only if the interval between job releases is constant. In our analysis, we assume that all tasks are periodic and share a single processor; however, our scheduling conditions also apply if tasks are *sporadic*, in which case a minimum separation (but no maximum separation) is assumed between job releases. We also assume that time is discrete, i.e., all release times, periods, and computation costs are integers. We say that a task *executes at time t* if it executes during the interval $[t, t+1)$. We implicitly assume that tasks share a set of objects implemented by using lock-free algorithms.

Note that there is no need to explicitly include such objects in our model, because operations on lock-free objects are implemented by task-level code sequences. For simplicity, we assume that jobs can be preempted at arbitrary points during their execution, and ignore system overheads like context switch times, interrupt handler overheads, etc. Techniques to account for system overheads have been described elsewhere [Katcher et al. 1993; Jeffay and Stone 1993], and such techniques can be applied to the scheduling conditions derived in this paper. (In fact, we employ the techniques of [Jeffay and Stone 1993] in analyzing the videoconferencing system described in Section 6.)

We say that a job is *interfered with* (or experiences an *interference*) if it executes a lock-free retry loop that does not successfully complete. For now, we assume that the *deadline* of a job of a task is the end of the corresponding period of that task. Later, at the end of Sections 3 and 4, we briefly consider scheduling conditions in which this assumption is relaxed. A task set is *schedulable* if and only if all jobs of all tasks meet their deadlines. The following is a list of symbols used in deriving our scheduling conditions.

— N - The number of tasks in the system. We use i and j as task indices. Unless stated otherwise, we assume that i and j are universally quantified over $\{1, \ldots, N\}$.
— T_i - The i^{th} task in the system.
— p_i - The period of task T_i. Tasks are sorted in nondecreasing order by their periods, i.e., $p_i < p_j \Rightarrow i < j$.
— $r_i(k)$ - The release time of the k^{th} job of T_i, where $r_i(k) = r_i(1) + (k-1) \cdot p_i$. We use k as a job index. Unless stated otherwise, we assume that k is universally quantified with range $k \geq 1$.
— $J_{i,k}$ - The k^{th} job of task T_i.
— c_i - The worst-case computational cost (execution time) of task T_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.
— s - The execution time required for one loop iteration in the implementation of a lock-free object, which for simplicity is assumed to be the same for all objects.

Under RM scheduling, we assume that if two tasks have the same period, then the task with the smaller task index has higher priority. Under EDF scheduling, we assume that if two jobs have the same deadline, then the job with the earlier release time has higher priority; if two such jobs are released at the same time, then the one with the smaller index has higher priority.

We obtain conditions for schedulability by determining the worst-case unfulfilled demand of each task. The *unfulfilled demand* of task T_i at time t is the remaining computation time of T_i's current job. The unfulfilled demand of T_i decreases by one from time t to time $t+1$ if a job of T_i executes at time t. When a job of task T_i is released, T_i's unfulfilled demand increases by c_i. Task T_i's unfulfilled demand can also increase due to interferences experienced by its jobs. Such increases are characterized by the following *interference assumptions*.

IA1 *A job J experiences an interference at time t if and only if, for some $t' \leq t$, (i) J executes at time $t'-1$, (ii) J is preempted at time t' by some higher-priority*

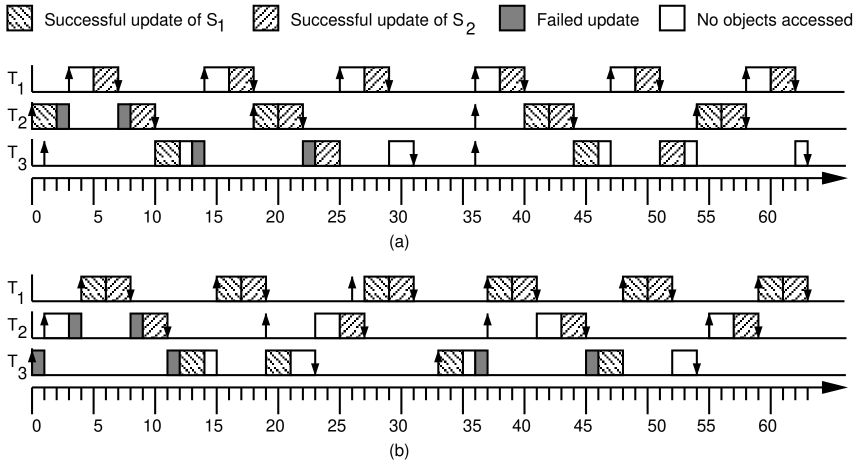

Fig. 2. Illustration of the task sets defined in Examples 1 and 2.

job, (iii) only higher-priority jobs execute in the interval $[t', t]$, *(iv) no higher-priority job that accesses an object in common with* J *is released in* $[t', t)$, *and (v) at least one such job is released at time* t. *(This implies that job* J *can be interfered with at most once during any interval when it is preempted.)*

IA2 *Each interference experienced by a job* $J_{i,k}$ *increases the unfulfilled demand of* T_i *by* s.

IA1 is pessimistic because the preempted job J may not, in fact, be accessing any shared object when preempted. IA2 is pessimistic because the cost of executing one iteration of the retry loop of any object is assumed to be s units. If retry loop costs are not the same, IA2 essentially requires that the unfulfilled demand of a task T_i be increased by the cost of the largest retry loop, for each interference in jobs of T_i.

In the proofs of our scheduling conditions, we also use the notion of task "demand", which is related to the notion of unfulfilled demand. The *demand* placed by a task T_i on the processor in an interval $[t, t']$ is the amount of processing time required by jobs of T_i in that interval [Jeffay et al. 1991]. In particular, task T_i's demand in $[t, t']$ includes T_i's unfulfilled demand at time t, c_i time units for each job release of T_i in $(t, t']$, and s time units for each interference occurring within $(t, t']$ in jobs of task T_i. A task is said to be *inactive* at time t if it places no demand on the processor at that time. The following examples illustrate some of the subtleties of our task model.

Example 1. Let a task T_i be given by the tuple $(r_i(1), c_i, p_i)$. Consider the following set of periodic tasks scheduled under the RM scheme.

$$T_1 = (3, 4, 11) \qquad T_2 = (0, 4, 18) \qquad T_3 = (1, 7, 35)$$

Assume that object S_1 is accessed by T_2 and T_3, and S_2 is accessed by T_1, T_2, and T_3. Tasks T_1 and T_2 access S_1 and S_2 in that order. Also, assume that $s = 2$.

The execution of the above task set is illustrated in Figure 2(a). In this figure, up-arrows represent job releases, down-arrows represent job completions, and shaded regions represent shared object accesses. Because the deadline of a job of a task corresponds to the release of the next job of that task, up-arrows also denote job deadlines.

We see that job $J_{3,1}$ experiences an interference at time 14, when job $J_{1,2}$ is released. This is because job $J_{1,2}$ is the earliest job that accesses an object in common with $J_{3,1}$ and that is released in the closed interval between $J_{3,1}$'s preemption at time 14 and its subsequent resumption at time 22. Thus, $J_{3,1}$'s operation on S_2 that begins at time 13 fails. This operation is retried at time 18, when it is successfully completed. Observe that $J_{3,1}$ experiences only one interference in the interval $[14, 22]$, even though two jobs ($J_{1,2}$ and $J_{2,2}$) are released in that interval that can potentially interfere with $J_{3,1}$'s object access. By IA1, we assume that $J_{2,4}$ interferes with $J_{3,2}$ at time 54 (because $J_{2,4}$ preempts $J_{3,2}$), even though $J_{3,2}$ is not actually accessing a shared object at that point. Thus, the upper bound on interference costs in the analysis presented later is rather pessimistic.

Task T_3's unfulfilled demand increases by 7 units at time 1 due to a job release and increases by 2 units at time 14 due to an interference by $J_{1,2}$. Also, task T_3's unfulfilled demand decreases by one unit at every instant in the interval $[11, 14]$ because it executes on the processor. The demand placed by T_3 on the processor in the interval $[5, 37]$ is 16 units; 7 units due to T_3's unfulfilled demand at time 5, 2 units due to an interference in $J_{3,1}$ at time 14, and 7 units due the release of $J_{3,2}$ at time 36.

Example 2. Consider the following set of periodic tasks scheduled under the EDF scheme.

$$T_1 = (4, 4, 11) \qquad T_2 = (1, 4, 18) \qquad T_3 = (0, 7, 33)$$

Assume that object S_1 is accessed by T_1 and T_3, and S_2 is accessed by T_1 and T_2. Task T_1 accesses S_1 and S_2 in that order; T_3 accesses S_1 twice. As in the previous example, we assume that $s = 2$.

The execution of the above task set is illustrated in Figure 2(*b*). We see that job $J_{3,1}$ experiences an interference at time 4 while accessing S_1, when job $J_{1,1}$ is released. $J_{1,1}$ has an earlier deadline than $J_{3,1}$ and is the first job that accesses an object in common with $J_{3,1}$ and that is released in the closed interval between $J_{3,1}$'s preemption at time 1 and its subsequent resumption at time 11. Observe that $J_{2,1}$ also experiences an interference at time 4 while accessing object S_2 due to the release of $J_{1,1}$. The total demand placed in the interval $[0, 25]$ by jobs with deadlines at or before 25 is 10 units; 4 and 6 units due to tasks T_1 and T_2, respectively. Of the 6 units of T_2's demand, 2 units are due to an interference in T_2 at time 4. Note that we do not include the demand due to jobs $J_{1,2}$, $J_{2,2}$ and $J_{3,1}$ because their deadlines are after time 25.

Before we present our scheduling conditions, we prove several lemmas used in the proofs of these conditions. In [Liu and Layland 1973], it is shown that for independent tasks (i.e., tasks that do not share objects), the longest response time of a task occurs at a *critical instant* of time, at which jobs of that task and all higher-priority tasks are released. However, this is not necessarily the case if tasks share lock-free objects, as illustrated in Example 1. In this example, the longest response time of task T_3 does not occur when its job is released along with higher-priority jobs at time 36. The job released at time 1 has a longer response time.

Instead of defining the critical instant, we introduce the notion of a "busy point". The *busy point* $b_i(k)$ of job $J_{i,k}$ is the latest point in time at or before $r_i(k)$ when jobs that have priority at least that of $J_{i,k}$ are either inactive or have a job release. For example, in Figure 2, the busy point of $J_{3,1}$ occurs at time 0, i.e., $b_3(1) = 0$. Because each task is either inactive or releases a job at time 0, $b_i(k)$ is well-defined for any i and k. Our scheduling conditions are obtained by inductively counting interferences over intervals of time. A busy point provides a convenient instant at which to start such an inductive argument, because tasks that are inactive or that have just released a job have experienced no interferences.

Lemma 1. *Consider any* $t \in [b_i(k), r_i(k+1))$ *at which* $J_{i,k}$ *has positive unfulfilled demand. Let* v *be the priority of* $J_{i,k}$. *In the interval* $[b_i(k), t]$, *the number of interferences in jobs with priority at least* v *is bounded by the number of instants in the interval* $(b_i(k), t]$ *at which some job with priority greater than* v *is released.*

Proof. To simplify the proof, we prove a slightly stronger statement: the number of interferences in the interval $[b_i(k), t]$ in jobs with priority at least v is bounded by the difference between (i) the number of instants in the interval $(b_i(k), t]$ at which some job with priority greater than v is released and (ii) the number of preempted jobs at time t that have not been interfered with[4] and that have priority at least v. The proof is by induction on t.

Basis: We show that the lemma holds at $b_i(k)$, $J_{i,k}$'s busy point. There can be no interferences in the interval $[b_i(k), b_i(k)]$ in jobs with priority at least v because $J_{i,k}$ and higher-priority jobs are either inactive or have a job release at $b_i(k)$. For the same reason, there are no preempted jobs at $b_i(k)$ with priority at least v. Clearly, there are zero instants in the interval $(b_i(k), b_i(k)]$ at which some job is released that has priority greater than v. Hence, the basis of the induction holds.

Induction Step: Assume that the above lemma holds at time $t - 1 \geq b_i(k)$. Let J be some job executing at time $t - 1$. (Such a job J must exist by the definition of $b_i(k)$.) Suppose that there are f interferences in the interval $[b_i(k), t-1]$ in jobs with priority at least v, and that there are w instants in the interval $(b_i(k), t-1]$ at which some job with priority greater than v is released. Also, suppose that there are x preempted jobs at time $t-1$ that have priority at least v and that have not been interfered with. Our inductive hypothesis can be formally written as $f \leq w - x$. We now consider two cases.

Case 1: If no job is released at time t that has priority greater than v, then by IA1, no interference can occur at that time in any job with priority at least v. Hence, there are f interferences in $[b_i(k), t]$ and w instants in $(b_i(k), t]$ at which some job is released that has priority greater than $J_{i,k}$'s priority. Also, the number of preempted jobs at time t is either $x - 1$ or x, depending on whether J completes at time t or not. In either case, the lemma holds at time t because our inductive hypothesis implies both $f \leq w - (x-1)$ and $f \leq w - x$.

Case 2: If $y > 0$ jobs are released at time t that have priority greater than v, then there are $w + 1$ instants in the interval $(b_i(k), t]$ at which some job is released

[4]Note that jobs that have been released but have not yet executed cannot be interfered with.

that has priority greater than v. Suppose that some number q of the x preempted jobs incur an interference at time t due to some newly released job that accesses a common object. We consider three subcases.

J has higher priority than all newly released jobs. It follows from IA1 that none of the jobs released at time t can interfere with J. Thus, there are $f+q$ interferences in $[b_i(k), t]$ and $x-q$ preempted jobs that have not been interfered with. (The y jobs released at time t cannot be interfered with because they have not started execution yet.) The lemma holds at time t because our inductive hypothesis implies $f+q \leq (w+1)-(x-q)$.

J is preempted at time t but none of the newly released jobs accesses an object in common with J. By IA1, none of the newly released jobs can interfere with J. Therefore, the number of interferences in $[b_i(k), t]$ is $f+q$. The number of preempted jobs that have not been interfered with is $x-q+1$ (including J). The lemma holds at time t because our inductive hypothesis implies $f+q \leq (w+1)-(x-q+1)$.

J is preempted at time t and some newly released job accesses an object in common with J. It follows from IA1 that J is interfered with at time t. Hence, the number of interferences in $[b_i(k), t]$ is $f+q+1$. The number of preempted jobs that have not been interfered with is $x-q$. Again, our inductive hypothesis implies $f+q+1 \leq (w+1)-(x-q)$. $\square$

Lemma 2. Consider any $t \in [b_i(k), r_i(k+1))$. Under the RM scheme, the number of interferences in T_i and higher-priority tasks in the interval $[b_i(k), t]$ is at most

$$\sum_{j=1}^{i-1} \left\lceil \frac{t-b_i(k)}{p_j} \right\rceil.$$

Proof. From Lemma 1, it follows that the number of interferences in jobs with priority at least that of $J_{i,k}$ in the interval $[b_i(k), t]$ is bounded by the number of instants in $(b_i(k), t]$ at which some job is released that has priority greater than that of $J_{i,k}$. Under the RM scheme, only jobs of tasks T_1 through T_{i-1} have priority greater than $J_{i,k}$, and the number of jobs of task T_j released in the interval $(b_i(k), t]$ is at most $\lceil (t-b_i(k))/p_j \rceil$. Therefore, the number of interferences in T_i and higher-priority tasks in the interval $[b_i(k), t]$ is bounded by $\sum_{j=1}^{i-1} \left\lceil \frac{t-b_i(k)}{p_j} \right\rceil$. $\square$

Lemma 3. Under the RM scheme, if, at time $r_i(k+1)-1$, T_i has positive unfulfilled demand and the total unfulfilled demand of T_i and higher-priority tasks is greater than one, then, for any t in the interval $[b_i(k), r_i(k+1))$, the difference between (i) the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$, and (ii) the available processor time in that interval, is greater than one.

Proof. The proof can be established by contradiction. To this end, suppose that there exists a $t \in [b_i(k), r_i(k+1))$ such that the difference between the total demand placed by tasks T_1 through T_i in the interval $[b_i(k), t]$ and the available processor time in that interval is at most one. It follows that the total unfulfilled demand of tasks T_1 through T_i at time t equals zero or one. Because the total unfulfilled demand of T_i and higher-priority tasks is greater than one at time $r_i(k+1)-1$, $t \neq r_i(k+1)-1$ holds. Also, either tasks T_1 through T_i are inactive at time t, or one of tasks T_1 through T_i has unit unfulfilled demand and is the highest-priority job executing on the processor. In both cases, it follows that T_i and higher-priority tasks are either inactive or have a job release at time $t+1$. To complete the proof, we show that this leads to a contradiction.

By the definition of a busy point, $b_i(k)$ is the latest time at or before $r_i(k)$ at which T_i and higher-priority tasks are either inactive or have a job release. Thus, it follows that $t+1$ cannot lie in the interval $(b_i(k), r_i(k)]$. Also, as explained earlier, $t \neq r_i(k+1) - 1$, i.e., $t+1 \neq r_i(k+1)$. Hence, $t+1$ lies in the interval $(r_i(k), r_i(k+1))$. T_i clearly cannot have a job release in the interval $[t+1, r_i(k+1))$ because $t+1 > r_i(k)$. Thus, T_i is inactive — and hence has no unfulfilled demand — throughout the interval $[t+1, r_i(k+1))$, contradicting our assumption that T_i has positive unfulfilled demand at time $r_i(k+1) - 1$. □

Lemma 4. *Under the EDF scheme, the number of interferences in jobs with a deadline at or before* $r_i(k+1)$ *in the interval* $[b_i(k), r_i(k+1))$ *is at most*

$$\sum_{j=1}^{N} \left\lfloor \frac{r_i(k+1) - b_i(k) - 2}{p_j} \right\rfloor .$$

Proof. From Lemma 1, it follows that the number of interferences in the interval $[b_i(k), r_i(k+1))$ is bounded by the number of instants in $(b_i(k), r_i(k+1))$ at which some job is released that has priority greater than $J_{i,k}$'s priority, i.e., the released job's deadline is before $r_i(k+1)$. Under the EDF scheme, the number of jobs of T_j released in the interval $(b_i(k), r_i(k+1))$ that have a deadline before $r_i(k+1)$ is at most $\left\lfloor \frac{(r_i(k+1)-1)-(b_i(k)+1)}{p_j} \right\rfloor$. Therefore, the number of interferences in jobs with deadlines at or before $r_i(k+1)$ in the interval $[b_i(k), r_i(k+1))$ is bounded by $\sum_{j=1}^{N} \left\lfloor \frac{r_i(k+1)-b_i(k)-2}{p_j} \right\rfloor$. □

3. STATIC-PRIORITY SCHEDULING CONDITIONS

In this section, we give separate necessary and sufficient conditions for the schedulability of a set of periodic tasks that share lock-free objects under the RM scheme. We also briefly consider the DM scheme. The following theorem gives a necessary scheduling condition for the RM scheme. The left-hand side of the quantified expression given below gives the minimum demand — which arises when there are no interferences — placed on the processor by T_i and higher-priority tasks in the interval $[0, t]$, where $0 < t \leq p_i$. The right-hand side gives the available processor time in that interval. For brevity, we omit the proof of this condition here, because it is similar to that given for independent tasks in [Lehoczky et al. 1989].[5] (A formal proof within our model can be found in [Anderson et al. 1995].)

Theorem 1. (*Necessity under RM*) *If set of periodic tasks that share lock-free objects is schedulable under the RM scheme, then the following condition holds for every task* T_i.

[5] Our necessary condition differs slightly from that in [Lehoczky et al. 1989] because we allow tasks to release their first jobs at arbitrary times.

$$\langle \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lfloor \frac{t}{p_j} \right\rfloor \cdot c_j \leq t \rangle$$

The next theorem gives a sufficient scheduling condition for the RM scheme. The left-hand side of the quantified expression given below gives the maximum demand placed by T_i and higher-priority tasks in the interval $[0, t)$. The first summation represents the demand placed on the processor by T_i and higher-priority tasks, not including the demand due to interferences. The second summation represents the total additional demand placed on the processor due to interferences in T_i and higher-priority tasks. The right-hand side of the expression is the available processor time in $[0, t)$. Observe that this condition can be applied without knowledge of which tasks access which objects.

Theorem 2. (*Sufficiency under RM*) *A set of periodic tasks that share lock-free objects is schedulable under the RM scheme if the following condition holds for every task* T_i.

$$\langle \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle$$

PROOF. We prove that if a task set is not schedulable, then the negation of the above expression holds. Assume that the given task set is not schedulable. Let $J_{i,k}$ be the first job to miss its deadline. (If several jobs simultaneously miss their deadline along with $J_{i,k}$, then let $J_{i,k}$ be the one with highest priority, i.e., smallest task index.) Consider any t in the interval $[b_i(k), r_i(k+1))$. We begin by deriving a bound on $D(b_i(k), t)$, the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$. $D(b_i(k), t)$ is comprised of the demand placed by job releases and the extra demand placed by interferences. Recall that at the busy point $b_i(k)$, T_i and all higher-priority tasks are either inactive or have a job release. Each job release of some task T_j introduces a demand of c_j on the processor, and there are at most $\lceil (t - b_i(k) + 1)/p_j \rceil$ job releases of that task in the interval $[b_i(k), t]$. Therefore, the total demand placed on the processor due to job releases of T_i and higher-priority tasks is at most $\sum_{j=1}^{i} \lceil (t - b_i(k) + 1)/p_j \rceil c_j$.

By Lemma 2, the number of interferences in jobs of T_i and higher-priority tasks in the interval $[b_i(k), t]$ is bounded by $\sum_{j=1}^{i-1} \lceil (t - b_i(k))/p_j \rceil$. By IA2, each interference introduces s units of additional demand on the processor. Therefore, the total additional demand due to interferences in jobs of T_i and higher-priority tasks is at most $\sum_{j=1}^{i-1} \lceil (t - b_i(k))/p_j \rceil s$. Therefore, we have

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

Job $J_{i,k}$ will miss its deadline if and only if, at time $r_i(k+1) - 1$, T_i has positive unfulfilled demand and the total unfulfilled demand of T_i and higher-priority tasks is greater than one. By Lemma 3, it follows that the difference between the total demand due to 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. Hence, we have the following.

$$D(b_i(k), t) - (t - b_i(k)) > 1$$

Using the bound on $D(b_i(k), t)$ derived above, the previous expression implies the following.

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

The above expression holds for all t in the interval $[b_i(k), r_i(k+1))$. Because this expression is independent of the end points (it is a function of the length of the interval), we can replace $t - b_i(k)$ with t', where $t' = t - b_i(k)$ and $t' \in [0, r_i(k+1) - b_i(k))$. Hence, we have the following.

$$\sum_{j=1}^{i} \left\lceil \frac{t' + 1}{p_j} \right\rceil c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t'}{p_j} \right\rceil s > t' + 1$$

Now, replace t' with t in the above expression, where $t = t' + 1$ and $t \in (0, r_i(k+1) - b_i(k)]$. Then, the following holds for all $t \in (0, r_i(k+1) - b_i(k)]$.

$$\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil s > t$$

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

In the videoconferencing system described in Section 6, tasks are sporadic rather than periodic, and job deadlines and release points do not necessarily coincide, as we have assumed. However, as remarked earlier, the scheduling conditions that we derive also apply if tasks are sporadic. In addition, the scheduling condition of Theorem 2 can be easily adapted to apply if job deadlines and release points do not coincide. This requires changing our model to allow the relative deadline l_i of task T_i to range over $(0, p_i]$ — by *relative deadline*, we mean the elapsed time between a job's release time and its deadline. For simplicity, we assume that tasks are indexed in nondecreasing order by relative deadline.

With this change to our model, it is possible to prove the following static-priority scheduling condition. This condition assumes that priority is assigned by the DM scheme [Leung and Whitehead 1982], in which tasks with smaller relative deadlines have higher priorities. The two summation terms in the stated expression below give the demand due to job releases of T_i and higher-priority tasks, and the additional demand due to interferences in those tasks, respectively, in an interval of length t.

Theorem 3. (*Sufficiency under DM*) *A set of periodic tasks that share lock-free objects is schedulable under the DM scheme if the following condition holds for every task T_i.*

$$\langle \exists t : 0 < t \leq l_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle$$

In comparing this condition to that given in Theorem 2, we see that t now ranges up to l_i, the relative deadline of T_i, rather than up to p_i, the period of T_i. Observe that when deadlines coincide with job releases, this condition reduces to the sufficient condition for RM scheduling proved in Theorem 2.

4. DYNAMIC-PRIORITY SCHEDULING CONDITIONS

In this section, we give separate necessary and sufficient conditions for the schedulability of a set of periodic tasks that share lock-free objects under the EDF scheme. The following theorem gives a necessary scheduling condition for the EDF scheme. According to this theorem, a task set is schedulable only if processor utilization is at most one. This condition is the same as that given in [Liu and Layland 1973] for independent tasks, so for brevity, its proof is omitted here. (A formal proof within our model is given in [Anderson et al. 1995].)

Theorem 4. (*Necessity under EDF*) *If set of periodic tasks that share lock-free objects is schedulable under the EDF scheme, then*

$$\sum_{i=1}^{N} \frac{c_i}{p_i} \leq 1.$$

The next theorem gives a sufficient condition for schedulability under the EDF scheme. It states that a task set is schedulable if processor utilization is at most one, when interferences are taken into account. Like the RM sufficiency condition of the previous section, this condition can be applied without knowledge of which tasks access which objects.

Theorem 5. (*Sufficiency under EDF*) *A set of periodic tasks that share lock-free objects is schedulable under the EDF scheme if the following condition holds.*

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

Proof. We prove that if a task set is not schedulable then $\sum_{j=1}^{N}(c_j + s)/p_j > 1$ holds. Assume that the given task set is not schedulable. Let $J_{i,k}$ be the first job to miss its deadline. (If several jobs simultaneously miss their deadline along with $J_{i,k}$, then let $J_{i,k}$ be the one with lowest priority.) We begin by deriving a bound on $D(b_i(k), r_i(k+1) - 1)$, the total demand placed on the processor by $J_{i,k}$ and higher-priority jobs, i.e., jobs with deadlines at or before $r_i(k+1)$, in the interval $[b_i(k), r_i(k+1))$. $D(b_i(k), r_i(k+1) - 1)$ is comprised of the demand placed by job releases and the extra demand placed by interferences. Recall that at $J_{i,k}$'s busy point, all jobs of equal or higher priority either are inactive or have a job release. Each job of some task T_j can place a demand of c_j on the processor, and there are at most $\lfloor (r_i(k+1) - b_i(k))/p_j \rfloor$ job releases of that task in the interval $[b_i(k), r_i(k+1))$

that have a deadline at or before $r_i(k+1)$. Therefore, the total demand placed on the processor due to such jobs is at most

$$\sum_{j=1}^{N} \left\lfloor \frac{r_i(k+1) - b_i(k)}{p_j} \right\rfloor \cdot c_j.$$

By Lemma 4, the total number of interferences in jobs with deadlines at or before $r_i(k+1)$ in the interval $[b_i(k), r_i(k+1))$ is bounded by the term $\sum_{j=1}^{N} \lfloor (r_i(k+1) - b_i(k) - 2)/p_j \rfloor$. By IA2, each interference requires s units of additional demand. Therefore, the total additional demand due to interferences is at most

$$\sum_{j=1}^{N} \left\lfloor \frac{r_i(k+1) - b_i(k) - 2}{p_j} \right\rfloor \cdot s.$$

Job $J_{i,k}$ will miss its deadline if and only if the difference between the total demand due to tasks with a deadline at or before $r_i(k+1)$ in the interval $[b_i(k), r_i(k+1))$ and the available processor time in that interval is greater than one. Therefore, we have the following.

$$\sum_{j=1}^{N} \left\lfloor \frac{r_i(k+1) - b_i(k)}{p_j} \right\rfloor \cdot c_j + \sum_{j=1}^{N} \left\lfloor \frac{r_i(k+1) - b_i(k) - 2}{p_j} \right\rfloor \cdot s - (r_i(k+1) - b_i(k) - 1) > 1$$

The above expression implies

$$\sum_{j=1}^{N} \frac{(r_i(k+1) - b_i(k)) \cdot c_j}{p_j} + \sum_{j=1}^{N} \frac{(r_i(k+1) - b_i(k) - 2) \cdot s}{p_j} > r_i(k+1) - b_i(k),$$

which implies the following.

$$\sum_{j=1}^{N} \frac{(r_i(k+1) - b_i(k)) \cdot (c_j + s)}{p_j} > r_i(k+1) - b_i(k)$$

Canceling out $r_i(k+1) - b_i(k)$ from both sides of the expression above yields the following expression, completing the proof.

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

□

When considering static-priority schemes in Section 3, we presented a condition that can be applied when task deadlines and periods do not coincide. The following theorem gives a similar result for EDF scheduling. (Recall that l_j is the relative deadline of task T_j.)

Theorem 6. (*Sufficiency under EDF when deadlines differ from periods*) *A set of periodic tasks that share lock-free objects is schedulable under the EDF scheme if the following condition holds.*

$$\langle \forall t : t \geq 0 : \sum_{j=1}^{N} \left\lfloor \frac{t - l_j + p_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=1}^{N} \left\lfloor \frac{t - 1 - l_j + p_j}{p_j} \right\rfloor \cdot s \leq t \rangle$$

As formulated above, the expression in Theorem 6 cannot be verified because the range of t is unbounded. However, there is an implicit bound on t. For example, if all tasks are released at time 0, then we only need to consider values less than or equal to the least common multiple of the task periods. If tasks are released at different times, an implicit bound still exists, but it is more complex [Baruah et al. 1993]. Also, if an upper bound on the utilization available for the tasks is known, then we can restrict t to a much smaller range [Baruah et al. 1993].

Observe that the summation terms in both Theorems 5 and 6 range from 1 to N. As a result, we are allowing for the fact that any of the N tasks can cause interferences in other tasks. However, by our task model, T_N cannot cause any interferences in other tasks. It is a straightforward exercise to tighten both conditions to account for this fact.

5. FORMAL COMPARISON

In this section, we compare lock-free objects to lock-based synchronization schemes and wait-free objects. This comparison is based upon the scheduling conditions presented in the previous two sections, and scheduling conditions for lock-based schemes found in the literature. For simplicity, we assume that all accesses to lock-based objects require r units of time, and that there are no nested object calls. (We reconsider the subject of nested calls later in Section 7.) Thus, the computation time c_i of a task T_i can be written as $c_i = u_i + m_i \cdot t_{acc}$, where u_i is the computation time not involving accesses to shared objects, m_i is the number of shared object accesses by T_i, and t_{acc} is the maximum computation time for any object access, i.e., s for lock-free objects and r for lock-based objects. (Recall that c_i is the computation time of T_i when it is the only task executing on the processor, i.e., it does not including blocking terms associated with priority inversions in the lock-based case or interference costs in the lock-free case.) As explained below, recent studies that evaluate the performance of lock-free objects [Massalin 1992] and lock-based objects [Gallmeister and Lanier 1991] indicate that s is likely to be less than r for many common objects. This is confirmed by the experimental results presented in Section 6.

5.1 Static-Priority Scheduling

We begin by comparing the overhead of lock-free object sharing under RM scheduling with the overhead of the lock-based priority ceiling protocol (PCP) [Rajkumar 1991]. When tasks synchronize by locking, a higher-priority job can be blocked by a lower-priority job that accesses a common object; the maximum blocking time is called the *blocking factor*. Under the PCP, the worst-case blocking time equals the time required to execute the longest critical section. Since we do not consider nested critical sections, the blocking factor equals r, the time to execute a single critical section. We denote the schedulability condition for periodic tasks using the PCP by the predicate *sched_PCP*, which on the basis of the analysis in [Rajkumar 1991], is defined as follows.

$$sched\_PCP \equiv \langle \forall i\ \exists t : 0 < t \leq p_i : r + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot r) \leq t \rangle$$

In the above equation, the first term on the left-hand side represents the blocking factor. In the second term, $u_j + m_j \cdot r$ represents the computation time of task T_j.

The expression on the left-hand side represents the maximum demand due to T_i and higher-priority tasks in a interval of length t.

We now derive conditions under which lock-free objects are guaranteed to perform at least as well as lock-based objects under the PCP. Consider the following derivation.

$$\langle \forall j : j \leq i : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \ \wedge \ \mathit{sched\_PCP}$$

$$\{\text{Substituting } (m_j + 1) \cdot s \text{ for } m_j \cdot r \text{ in } \mathit{sched\_PCP}\}$$

$$\Rightarrow \langle \forall i \ \exists t : 0 < t \leq p_i : \textstyle\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot s) + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot s + r \leq t \rangle \quad (1)$$

$$\Rightarrow \langle \forall i \ \exists t : 0 < t \leq p_i : \textstyle\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot s) + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle \quad (2)$$

Because $c_j = u_j + m_j \cdot s$ in the lock-free case, the last expression in this derivation is equivalent to the scheduling condition of Theorem 2. Note that $s \leq \frac{r}{2}$ implies that $\langle \forall j : j \leq i : (m_j + 1) \cdot s \leq m_j \cdot r \rangle$ because, for positive m_j, $\frac{1}{2} \leq \frac{m_j}{m_j+1} < 1$. Thus, if the time taken to execute one iteration of a lock-free retry loop is less than half the time it takes to access a lock-based object under the PCP, then any task set that is schedulable under the PCP is also schedulable when using lock-free objects. This also implies that there are certain task sets that are schedulable when lock-free objects are used, but not under the PCP.

Since the above comparison between lock-free objects and the PCP rests on r and s values, it is instructive to take a closer look at what these values are actually comprised of in practice. s is the cost of one lock-free retry loop. For most simple data structures like read/write buffers, queues, and stacks, s is often simply the cost of a simple, straight-line code sequence. For more complex data structures like linked lists and balanced trees, a lock-free implementation would be more complicated, and corresponding s values might be relatively high. In some applications, s might also include the time taken to copy the implemented object if a universal construction were being used, or the time taken to simulate a synchronization primitive such as CAS2 if such primitives were not provided in hardware. Under the PCP, r includes the cost of a system call to modify the calling task's priority before accessing an object, the cost of actually performing the shared-object operation, and the cost of a system call to restore the calling task's priority after an access.

What are typical values of s and r? A performance comparison of various lock-free objects is given by Massalin in [Massalin 1992]. Massalin reports that, given hardware support for primitives like compare-and-swap, s varies from 1.3 microseconds for a counter to 3.3 microseconds for a circular queue. In the absence of hardware support, such primitives can be simulated by a trap, adding an additional 4.2 microseconds. Massalin's conclusions are based on experiments run on a 25 MHz, one-wait-state memory, cold-cache 68030 CPU. In contrast, lock-based implementations fared much worse in a recent performance comparison of commercial real-time operating systems run on a 25 MHz, zero-wait-state memory 80386 CPU [Gallmeister and Lanier 1991]. In this comparison, the implementation of semaphores on LynxOS took 154.4 microseconds to lock and unlock a semaphore in the worst case. The corresponding figure for POSIX mutex-style semaphores was

243.6 microseconds. Although these figures cannot be regarded as definitive, they do give some indication as to the added overhead when operating-system-based locking mechanisms are used. For the videoconferencing system described in Section 6, the situation is very similar. In this system, s is 37 time units, while r is 151 time units.

In deriving (2) from (1) in the comparison above, we dropped the term r, effectively ignoring the effect of blocking under the PCP. If blocking times are considerable, then lock-free objects would be more competitive than this comparison indicates. It should also be noted that our scheduling analysis is very pessimistic. In reality, a preempted task need not be accessing a shared object, and hence may not necessarily have an interference as we have assumed.

5.2 Dynamic-Priority Scheduling

We now compare the overhead of lock-free objects with the dynamic deadline modification (DDM) scheme under EDF scheduling (EDF/DDM) [Jeffay 1992], which is a lock-based protocol for dynamic-priority schemes. Under this scheme, tasks are divided into one or more phases. During each phase, a task accesses at most one shared resource. Before a task T_i accesses a shared object S_m, its deadline is modified to the deadline of some task T_j that accesses S_m and that has the smallest deadline of all tasks that access S_m. Upon completing the shared object access, T_i's deadline is restored to its original value. In our comparison, we assume that phases in which some shared object is accessed are r units in length. Under the EDF/DDM scheme, r includes the cost of a system call to modify the task deadline before accessing an object, the cost of performing the shared-object operation, and the cost of a system call to restore the task deadline after an access. Based on the analysis of [Jeffay 1992], a sufficient condition for the schedulability of a set of periodic tasks under the EDF/DDM scheme, $sched\_DDM$, can be defined as follows.

$$
\begin{aligned}
sched\_DDM \equiv & \left(\textstyle\sum_{j=1}^{N} \frac{u_j + m_j \cdot r}{p_j} \le 1\right) \wedge \\
& \langle \forall i, t : P_i < t < p_i : r + \textstyle\sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{p_j} \right\rfloor \cdot (u_j + m_j \cdot r) \le t \rangle
\end{aligned}
$$

In the first conjunct of the above equation, the expression in the left-hand side of the inequality represents the total processor utilization due to tasks in the system. The term $u_j + m_j \cdot r$ represents the computation time of T_j. In the second conjunct, the term P_i is defined as the minimum p_j such that T_j shares a common object with T_i. The first term on the left-hand side of the inequality in the second conjunct represents the maximum time T_i can block tasks with smaller periods, and the second term represents the total demand on the processor due to tasks with smaller periods in an interval of length $t - 1$. The expression on the left-hand side of the inequality represents the maximum demand that can be placed on the processor during an interval of length t.

We now derive conditions under which lock-free objects are guaranteed to perform at least as well as objects implemented using the DDM scheme. Consider the following derivation.

$$
\langle \forall j : (m_j + 1) \cdot s \le m_j \cdot r \rangle \ \wedge \ sched\_DDM \tag{3}
$$

$$\{\text{By the definition of } sched\_DDM\}$$
$$\Rightarrow \langle \forall j : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \ \wedge \ \textstyle\sum_{j=1}^{N} \frac{u_j + m_j \cdot r}{p_j} \leq 1 \qquad (4)$$
$$\{\text{Substituting } (m_j + 1) \cdot s \text{ for } m_j \cdot r\}$$
$$\Rightarrow \textstyle\sum_{j=1}^{N} \frac{u_j + (m_j+1) \cdot s}{p_j} \leq 1$$

Because $c_j = u_j + m_j \cdot s$ in the lock-free case, the last expression in this derivation is equivalent to the scheduling condition of Theorem 5. As noted previously, $s \leq \frac{r}{2}$ implies $\langle \forall j : (m_j + 1) \cdot s \leq m_j \cdot r \rangle$. Thus, as with the PCP, if the time taken to execute one iteration of a lock-free retry loop is less than half the time it takes to access an object using the DDM scheme, then any task that is schedulable under the EDF/DDM scheme is also schedulable under EDF scheduling using lock-free objects. As mentioned previously, s is likely to be smaller than r for many objects.

In deriving (4) from (3) in the above comparison, we dropped the second conjunct in *sched_DDM*, effectively ignoring the effect of blocking under the EDF/DDM scheme. If blocking times are considerable, then lock-free objects would perform better than as indicated above.

5.3 Wait-Free Objects

Wait-free shared objects differ from lock-free objects in that wait-free objects are required to guarantee that individual tasks are free from starvation. Most wait-free universal constructions ensure termination by requiring each task to "help" every other task to complete any pending object access [Herlihy 1991; Herlihy 1993]. To see how this works, consider the lock-free universal construction of Herlihy [Herlihy 1993], which is described in Section 1. This construction does not guarantee termination because the *store-conditional* operation of each retry loop iteration may fail. Herlihy extends this construction to be wait-free by requiring each task to "announce" any pending operation by recording it in a shared array. Using this information, each task is able to "help" other tasks with pending operations by performing their operations in addition to its own. If a task is repeatedly unsuccessful in modifying the shared object pointer, then it is eventually helped by another task — in fact, after at most two retry loop iterations.

Note, however, that on a uniprocessor, lower-priority tasks cannot help higher-priority tasks because a higher-priority task does not release the processor until its demand has been fulfilled. Thus, each task only helps lower-priority tasks. Hence, the greater the task priority, the larger the access time. In some sense, the problem of priority inversion still exists, because a medium-priority task will have to wait while a high-priority task helps a low-priority task. On the other hand, when lock-free objects are used, the time to complete an object access decreases with increasing priority. For these reasons, some task sets that are schedulable when using lock-free objects will not be schedulable when using wait-free objects. This is true of the task set evaluated in the next section.

In order to more formally compare lock-free and wait-free objects, let us assume that objects are implemented using Herlihy's universal constructions. First, note that tasks that share wait-free objects can be viewed as independent tasks, i.e., the scheduling conditions derived in [Lehoczky et al. 1989] and [Liu and Layland

1973] apply. These conditions are the same as those given in Theorems 2 and 5, respectively, when $s = 0$. The computational cost c_j in this case equals $u_j + m_j \cdot t_{wf}$, where t_{wf} is the worst-case access time of a wait-free object, which occurs when *all* lower-priority tasks with pending operations are helped. If s is the time taken for the retry loop in Herlihy's less-complicated lock-free construction, then we would expect $t_{wf} = c \cdot s$, for some $c \gg 1$. Observe that if $c \geq 2$, then the c_j term in the wait-free case is greater than or equal to $u_j + (m_j + 1) \cdot s$. Note also that $u_j + (m_j + 1) \cdot s$ is at least as large as the terms corresponding to T_j in Theorems 2 and 5. Thus, if a task set is schedulable using Herlihy's wait-free universal construction, then it is also schedulable using Herlihy's lock-free universal construction.

The conclusion drawn above that lock-free implementations always perform better than wait-free ones may not apply if wait-free objects are implemented using techniques other than a Herlihy-like helping scheme. In fact, Anderson, Ramamurthy, and Jain have shown in a recent paper that it is possible to dramatically reduce helping overhead in wait-free implementations for priority-based real-time systems [Anderson et al. 1997]. For such implementations, it may indeed be the case that wait-free implementations are superior to lock-free ones for certain objects, although the exact extent (if any) to which this is true is currently unknown.

6. EXPERIMENTAL COMPARISON

In this section, we provide empirical evidence that lock-free objects are often superior to more traditional lock-based approaches to real-time object sharing. This evidence comes from a set of experimental comparisons performed using a real-time desktop videoconferencing system implemented at UNC [Jeffay et al. 1992]. We modified this system to support lock-free shared objects implemented under both DM and EDF scheduling, semaphores implemented using the PCP under DM scheduling, and semaphores implemented under EDF/DDM scheduling. We also considered wait-free shared objects implemented under both DM and EDF scheduling. The formal analysis for each synchronization scheme was applied to determine whether it was theoretically possible to ensure that no deadlines would be missed. We then executed the system using each synchronization scheme under a variety of loading conditions, and compared the actual performance to that predicted by the formal analysis. In all cases, the formal analysis predicted the actual behavior of the system. Moreover, our lock-free synchronization schemes frequently led to higher levels of sustainable system utilization than was possible with lock-based synchronization. Also, our experiments confirm that, for the objects considered, lock-free implementations are superior to wait-free implementations based on Herlihy-like helping schemes, for real-time computing on uniprocessors. The following subsection describes the videoconferencing system in more detail.

6.1 Experimental Setup

The videoconferencing system considered in our investigations acquires analog audio and video samples on a workstation and then digitizes, compresses, and transmits the samples over a local-area network to a second workstation where they are decompressed and displayed. Here we consider only the portion of the system responsible for the acquisition, compression, and network transmission of media samples by the sending workstation.

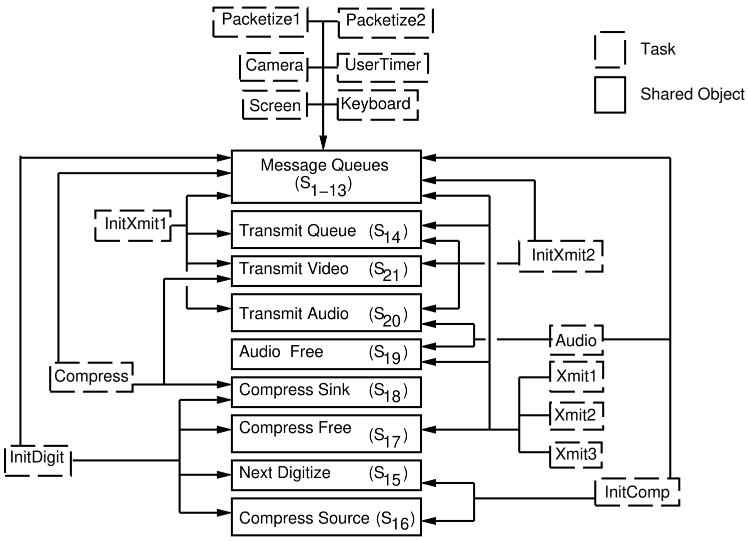

Fig. 3. Tasks and shared queues in the videoconferencing system.

Abstractly, the tasks on the sending workstation are organized as a software pipeline. Communication between stages is realized through a queue of media samples that is shared using a simple producer/consumer protocol. These queues must support a *get_length* operation in addition to *enqueue* and *dequeue*, which slightly complicates their implementation. Queues of shared media samples exist between the digitizing task and the compression task and between the compression task and the network transmission tasks. The real-time constraints on the operation of the pipeline require media samples to flow through the pipeline in a predictable manner. These media samples arrive sporadically and are manipulated by a set of sporadic tasks. Each task must process arriving media samples before a prespecified deadline that does not coincide with that task's period, and no media samples may be lost due to buffer overflows.

A comprehensive view of the tasks (dashed boxes) and shared queues (solid boxes) on the sending workstation is given in Figure 3. In this figure, an arrow is directed from each task to each of the shared objects it accesses. The message queues ($S_1 - S_{13}$) are used for inter-task communication. For our purposes, it suffices to consider the tasks in Figure 3 to be an abstract set of tasks — details regarding the function of each task, and how the tasks interact are not important to us. For a more detailed description of this system, we refer the interested reader to [Stone 1995].

We evaluated the performance of the system when the shared queues were implemented using lock-free algorithms, wait-free algorithms, and lock-based techniques. We implemented lock-free queues by using the shared queue implementation given in [Massalin 1992] (modified to support the *get_length* operation), and wait-free queues by using the wait-free universal construction given in [Herlihy 1993]. Massalin's queue implementation requires CAS (needed for the dequeue operation) and CAS2 (needed for the enqueue operation), and Herlihy's construction requires *load-*

linked and *store-conditional*. We implemented these primitives by short kernel calls; interrupts were disabled for the duration of these calls.

We found that the videoconferencing task set was not schedulable, in any experiment, when the shared queues were implemented using Herlihy's wait-free universal construction. This is due to the high overhead of helping, as discussed in Section 5.3. In contrast, our lock-free implementations required very little overhead, with interferences occurring only rarely. For example, in ten executions of the system, only 363 interferences occurred in 415,229 enqueue operations. We also found that multiple interferences of a single operation *never* occurred. In the following two subsections, we discuss results of experiments that were conducted to compare lock-free and lock-based schemes under static- and dynamic-priority scheduling.

6.2 Static-Priority Scheduling

In this subsection, we discuss the results of experiments that compare the overhead of lock-free objects to lock-based objects implemented using the PCP. In both cases scheduling was performed using the DM scheduling algorithm [Leung and Whitehead 1982].

Qualitatively, when queue synchronization was achieved using semaphores, approximately seven media samples were lost in the pipeline every second due to buffer overflow. In contrast, no media samples were lost when lock-free objects were used.

This result is predicted by the formal analysis of the system, which we now present. The model we consider consists of a set of $N = 15$ sporadic tasks, $M = 21$ shared objects, and $Q = 12$ periodic and sporadic interrupt handlers. The i^{th} task in the system is given by the tuple $\langle c_i, p_i, l_i, a_i \rangle$, where c_i and p_i have the usual meanings, l_i is the relative deadline of T_i, and a_i is the set of shared objects accessed by T_i. We assume that tasks are indexed in the order of nondecreasing deadlines. The i^{th} interrupt handler is given by the tuple $\langle e_i, v_i \rangle$, where e_i is the execution time of the handler and v_i is the minimum time between interrupts. Interrupt handlers are executed in a first-come-first-served manner and always have priority over application tasks. The periods, relative deadlines, and the execution times of the tasks in our formal model are shown in Table 2. The periods and execution times of the interrupt handlers are shown in Table 3.

The formal model of the experimental system can be analyzed by using the scheduling condition given in Theorem 3 when lock-free objects are used, and that given in [Lehoczky et al. 1989] when lock-based objects are used. Note, however, that these conditions do not consider the cost of handling interrupts, and hence cannot be used directly. Fortunately, this problem can be overcome by using techniques derived in [Jeffay and Stone 1993]. The idea is to derive an expression that bounds the demand due to interrupt handlers in any given interval, and to then account for this demand in the scheduling conditions of Theorem 3 and [Lehoczky et al. 1989].

Informally, we account for the cost of interrupt handlers as follows (see [Jeffay and Stone 1993] for a more formal version of this argument). First, we define the term $F(t)$ to be the cost of handling interrupts over an interval of length t. In order to derive a bound on $F(t)$, consider I_i, the i^{th} interrupt in the system, and consider an interval $[t_0, t_0 + t)$ of length t, where $t_0 \geq 0$. I_i occurs at most $\lceil t/v_i \rceil$ times in that interval, and requires e_i units of processor time for every occurrence. Hence,

Table 2. Task characteristics. Times are given in microseconds.

Task Name	T_i	Cost [DM] c_i		Cost [EDF] c_i		Period p_i	Deadline l_i	WC Response†	
		PCP	LF	DDM	LF			t_i^*	t_i^{**}
`InitXmit1`	T_1	579	459	687	649	33333	6705	4743	4623
`Xmit1`	T_2	147	147	147	147	45603	6705	4890	4807
`Xmit2`	T_3	147	147	147	147	45603	6705	5037	4991
`Xmit3`	T_4	147	147	147	147	45603	6705	5184	5175
`Compress`	T_5	602	528	669	624	9573	8000	5786	5740
`Camera`	T_6	396	396	416	416	15746	15000	6182	6173
`Audio`	T_7	1017	953	1024	966	15746	15000	7199	7163
`InitDigit`	T_8	1110	1046	1137	1096	31492	15000	8309	8246
`InitComp`	T_9	1332	746	1069	640	31492	15000	10243	9029
`InitXmit2`	T_{10}	710	604	821	982	33333	19850	11287	10235
`Packetize1`	T_{11}	8315	8315	8315	8315	40842	33333	22651	21943
`Packetize2`	T_{12}	8315	8315	8315	8315	40842	33333	N/A	30860
`UserTimer`	T_{13}	126	122	102	137	54538	54538	37872	31385
`Keyboard`	T_{14}	580	549	637	589	490853	490853	39054	37065
`Screen`	T_{15}	142	71	148	78	1963379	1963379	39196	37173

† Worst-case (WC) response times apply to DM scheduling.

Table 3. Interrupt handler execution times and periods. Times are given in microseconds.

I_i	I_1	I_2	I_3	I_{4-5}	I_{6-7}	I_{8-9}	I_{10-12}
e_i	254	333	333	183	389	389	389
v_i	54925	16666	10493	15492	45666	42603	47666

the total demand placed on the processor by I_i in the interval is at most $\lceil t/v_i \rceil \cdot e_i$. It then follows that the total demand due to all the interrupt handlers, $F(t)$, is bounded by the summation on the right-hand side of the following inequality.

$$F(t) \leq \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \tag{5}$$

Using (5), we can obtain a schedulability condition when the tasks synchronize using lock-based objects and the PCP. This involves modifying the condition presented in [Lehoczky et al. 1989] to account for the demand placed by interrupt handlers, as given by (5). The resulting condition is as follows.

$$\langle \forall i\ \exists t : 0 < t \leq l_i :\ r + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle \tag{6}$$

The first term in (6) gives the worst-case blocking time in the system, and the second term gives the demand placed by T_i and higher-priority tasks on the processor. The third term gives the maximum demand placed by all interrupt handlers in the same interval.

In our system, r equals 151. In Table 2, t_i^* gives a value of t in the interval $(0, l_i]$ that satisfies (6). The analysis shows that the task `Packetize2` is not schedulable. This task copies compressed media sample buffers to the network adapter. When `Packetize2` does not meet its deadline, the sender drops (never transmits) some of the media samples. This analysis explains why some media samples were lost when the system was run using lock-based objects and the PCP.

We now consider the system when the tasks synchronize using lock-free objects. A schedulability condition for this case is obtained by modifying the condition of Theorem 3 to account for the demand placed by interrupt handlers, as given by (5). The resulting condition is as follows.

$$\langle \forall i\ \exists t : 0 < t \leq l_i :\ \textstyle\sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle \quad (7)$$

In the above equation, the first term gives the demand placed on the processor due to T_i and higher-priority tasks. The second term gives the additional demand due to interferences, and the third term gives the maximum demand placed on the processor by interrupt handlers in the same interval. In our system, s equals 37. (Observe that s is less than $r/2$ in our system.) In Table 2, t_i^{**} gives a value of t in the interval $(0, l_i]$ that satisfies (7). It can be seen that all tasks are schedulable when lock-free objects are used. This is confirmed by the fact that no media samples are lost during the execution of the system.

6.3 Dynamic-Priority Scheduling

In this subsection, we discuss the results of experiments that compare the overhead of lock-free objects under the EDF scheme to lock-based objects under the EDF/DDM scheme. Our experiments showed that the task set of Tables 2 and 3 is schedulable under both schemes. This is result is predicted by the formal analysis of the system, which we now present.

Our analysis of the EDF/DDM scheme is based upon the following scheduling condition, which is proved in [Stone 1995].

$$(\textstyle\sum_{j=1}^{N} c_j/p_j + \sum_{j=1}^{Q} e_j/v_j \leq 1) \wedge$$
$$\langle \forall t : p_1 \leq t \leq B_{DDM} : \textstyle\sum_{j=1}^{N} \left\lfloor \frac{t-l_j+p_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle \wedge$$
$$\langle \forall i, t : p_1 < t < p_i : r + \textstyle\sum_{j=1}^{i-1} \left\lfloor \frac{t-1-l_j+p_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle$$

In the second conjunct above, $B_{DDM} \equiv (\sum_{j=1}^{N} c_j + \sum_{j=1}^{Q} e_j)/(1 - \sum_{j=1}^{N} c_j/p_j + \sum_{j=1}^{Q} e_j/v_j)$. The first and third conjuncts above correspond to the two conjuncts of *sched_DDM* given in Section 5.2. However, these conjuncts have been modified to account for the overhead of interrupt handlers, and to reflect the fact that in the videoconferencing system deadlines and job releases do not necessarily coincide. In the left-hand side of the inequality in the second conjunct, the first and second summation terms give the maximum demand due to the tasks and interrupt handlers, respectively, in an interval of length t. The right-hand side of the inequality gives the available processor time in that interval. It can be shown that this scheduling condition holds for the abstract task set defined in Tables 2 and 3.

Our analysis of the system when lock-free objects are used is based upon the scheduling condition below. This condition is based upon the conditions given in Theorems 5 and 6 and the techniques given in [Jeffay and Stone 1993] for accounting for the overhead of interrupt handlers.

$$(\textstyle\sum_{j=1}^{N} (c_j + s)/p_j + \sum_{j=1}^{Q} e_j/v_j \leq 1) \ \wedge$$

$\langle \forall t : t \in [p_1, B_{LF}] : \sum_{j=1}^{N} \left(\left\lfloor \frac{t-l_j+p_j}{p_j} \right\rfloor \cdot c_j + \left\lfloor \frac{t-1-l_j+p_j}{p_j} \right\rfloor \cdot s \right) + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle$

In this expression, $B_{LF} \equiv (\sum_{j=1}^{N}(c_j + s) + \sum_{j=1}^{Q} e_j)/(1 - \sum_{j=1}^{N}(c_j + s)/p_j + \sum_{j=1}^{Q} e_j/v_j)$. The first conjunct above is the condition of Theorem 5 augmented to include utilization due to interrupt handlers. The second conjunct follows from Theorem 6 and results of [Jeffay and Stone 1993]. The three summation terms in this conjunct give the maximum demand due to the tasks, interferences, and interrupt handlers, respectively, in an interval of length t. The right-hand side of the stated inequality gives the available processor time in that interval. It can be shown that this scheduling condition holds for the abstract task set defined in Tables 2 and 3.

In order to more precisely compare lock-free objects with objects implemented under the EDF/DDM scheme, we introduced a dummy task T_{16}, given by the tuple $\langle c_{16}, 2342664, 2342664, \{S_{17-21}\}\rangle$, to increase the processor utilization of the system. This dummy task consists of a bounded loop. During each loop iteration, the task performs some busy work and accesses some shared objects. The demand on the processor was varied by modifying the number of loop iterations executed by the dummy task.

Our experiments showed that processor utilization was higher under the EDF/DDM scheme for all task loads. Under the EDF/DDM scheme, tasks started to miss deadlines when the dummy task performed approximately 3500 loop iterations. The processor utilization corresponding to this load was close to 99.4%. For the same load, the processor utilization was only 94% when lock-free objects were used. Processor utilization is higher under EDF/DDM for the same load due to the overhead of modifying task deadlines for each shared object access. This confirms the prediction of Section 5.2 that lock-free objects often require less overhead than object implemented under the EDF/DDM scheme. In our experiments, when lock-free objects were used, tasks started missing deadlines when processor utilization was about 99.1%.

7. CONCLUDING REMARKS

Our results show that lock-free objects have a number of advantages over lock-based schemes such as the PCP for real-time computing on uniprocessors. First, lock-free objects can be applied without detailed knowledge of which tasks access which objects. Second, systems using lock-free objects can be easily modified to add tasks dynamically, since operating system tables do not have to be recomputed. Third, and most importantly, the use of lock-free objects often results in less overhead and lower task response times as compared to objects implemented using lock-based techniques.

It can be shown that the upper bound on the number of interferences we derived can be reduced if the first job of every task is released at the same time, and if the period of each task is a multiple of the smallest period. The intuitive reasoning for this observation is as follows. Consider two jobs J and J' that access a common object, where J' has a higher priority than J. If both jobs are released together, then J does not start executing before J' completes its execution, and hence J' cannot cause an interference in J. On the other hand, if J' is released at some time during J's execution, then it can cause an interference in J. Hence, by releasing the tasks together and by defining their periods to be multiples of the smallest period, we increase the number of instances at which jobs are released together and decrease the number of interferences. Moreover, it can be shown in this case that the number of interferences in jobs with priority greater than or equal to v in the interval $(t, t']$ is bounded by $\lceil (t' - t + 1)/p_1 \rceil$, which implies that the number of interferences of a task is independent of the level of that task.

Even in the absence of hardware support for primitives like CAS2 (refer to Figure 1), lock-free shared objects can be implemented with low overhead. On a uniprocessor, this can be achieved by a nonpreemptable kernel call that simulates the required primitive. This requires the introduction of a blocking factor y in our scheduling conditions. This nonpreemptable code fragment is smaller than one iteration of a lock-free retry loop, i.e., $y < s$. Observe that the introduction of this blocking term in our RM scheduling condition does not affect our comparison with the PCP because in comparing the two schemes, we ignored the blocking factor in *sched_PCP*. This reasoning also holds for the EDF/DDM scheme, because our comparison with that scheme ignored the second conjunct of *sched_DDM*, which includes the blocking factor under EDF/DDM scheduling. Even without kernel support for disabling preemptions, it is possible to efficiently simulate synchronization primitives like CAS and CAS2. This follows from recent work on wait-free implementations for priority-based real-time uniprocessor systems [Ramamurthy et al. 1996; Anderson and Ramamurthy 1996; Anderson et al. 1997]. This work shows that in such systems needed primitives can be simulated in a wait-free manner from other instructions (even reads and writes) entirely at the user level.

One advantage of lock-based schemes is that they allow critical sections to be arbitrarily nested. It might be useful, for example, to nest two critical sections to transfer the contents of one shared buffer to another. Together with Mark Moir, we recently developed a transaction-based framework that can provide this kind of functionality [Anderson et al. 1997]. Using this framework, a buffer transfer can be accomplished in a lock-free manner by performing a lock-free transaction that accesses both buffers. Our scheduling conditions are still applicable if multi-object accesses are allowed, provided s is defined to be the time taken by the longest retry loop. In this case, the longest loop would presumably be one that accesses several objects at once. If multi-object accesses are rare, then this could lead to scheduling conditions that are somewhat pessimistic. Although lock-free transactions are a promising idea, further experimentation is needed to determine if they can be effectively applied in real-time applications.

The task model considered in this paper assumes that all objects have equal access times. This is reasonable if all access times are comparable. Recent work by the first two authors has shown that when large variations in retry loop costs exist, tighter scheduling conditions can be obtained by using linear programming to obtain a better estimate of the cost associated with interferences [Anderson and Ramamurthy 1996]. 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 E involving a set of variables. Each variable represents the number of interferences of a particular lock-free loop in $\mathcal{I}$. The variables are constrained by a set of inequalities. 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, E

is maximized using linear programming. The result is a bound on interference costs. Simulation experiments reported in [Anderson and Ramamurthy 1996] indicate that when linear programming is used to bound interference costs, lock-free objects are likely to perform better than lock-based schemes if the average cost of a lock-free retry loop is at most the average cost of a lock-based access. By contrast, the results obtained in Section 5 indicate that when the conditions of this paper are applied, lock-free objects will perform better if the *worst-case* cost of a lock-free retry loop (s) is at most half the *worst-case* cost of a lock-based access (r).

ACKNOWLEDGMENTS

We are grateful to Rich Gerber and Ted Johnson for their comments on this paper. We also thank Dave Bennett, Don Stone, and Terry Talley for helping with the experimental work described in Section 6. We also thank the anonymous referees for their helpful comments.

REFERENCES

Alemany, J. and Felten, E. W. 1992. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing* (August 1992), pp. 125–134. ACM.

Anderson, J. H. and Moir, M. 1995. Universal constructions for large objects. In *Proceedings of the Ninth International Workshop on Distributed Algorithms* (September 1995), pp. 168–182. Springer Verlag.

Anderson, J. H. and Ramamurthy, S. 1996. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium* (December 1996), pp. 92–105. IEEE.

Anderson, J. H., Ramamurthy, S., and Jain, R. 1997. Implementing wait-free objects in priority-based systems. Manuscript.

Anderson, J. H., Ramamurthy, S., and Jeffay, K. 1995. Real-time computing with lock-free objects. Technical Report TR95-021, Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina.

Anderson, J. H., Ramamurthy, S., Moir, M., and Jeffay, K. 1997. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*. Amsterdam: Kluwer Academic Publishers.

Baker, T. 1991. Stack-based scheduling of real-time processes. *Real-Time Systems 3*, 1 (March), 67–99.

Baruah, S. K., Howell, R. R., and Rosier, L. E. 1993. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science 118*, 3–20.

Bershad, B. 1993. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (May 1993), pp. 264–274.

Chen, M. I. and Lin, K. J. 1990. Dynamic priority ceiling: A concurrency control protocol for real time systems. *Real-Time Systems 2*, 1, 325–346.

Gallmeister, B. O. and Lanier, C. 1991. Early experience with POSIX 1003.4 and POSIX 1003.4a. In *Proceedings of the 12th IEEE Real-Time Systems Symposium* (December 1991), pp. 190–198. IEEE.

Herlihy, M. 1991. Wait-free synchronization. *ACM Trans. on Programm. Lang. Syst. 13*, 1, 124–149.

Herlihy, M. 1993. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programm. Lang. Syst. 15*, 5, 745–770.

Jeffay, K. 1992. Scheduling sporadic tasks with shared resources in hard real-time systems. In *Proceedings of the* 13^{th} *IEEE Symposium on Real-Time Systems* (December 1992), pp.

89–98. IEEE.

Jeffay, K., Stanat, D. F., and Martel, C. U. 1991. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the* 12^{th} *IEEE Symposium on Real-Time Systems* (December 1991), pp. 129–139. IEEE.

Jeffay, K. and Stone, D. 1993. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the* 14^{th} *IEEE Symposium on Real-Time Systems* (December 1993), pp. 212–221. IEEE.

Jeffay, K., Stone, D., and Smith, F. D. 1992. Kernel support for live digital audio and video. *Computer Communications 15*, 6 (July), 388–395.

Johnson, T. and Harathi, K. 1994. Interruptible critical sections. Technical Report TR94-007, University of Florida, Gainesville, Florida.

Katcher, D. I., Arakawa, H., and Strosnider, J. K. 1993. Engineering and analysis of fixed-priority schedulers. *IEEE Trans. on Software Engineering 19*, 9 (September), 920–934.

Kopetz, H. and Reisinger, J. 1993. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Proceedings of the* 14^{th} *IEEE Symposium on Real-Time Systems* (December 1993), pp. 131–137. IEEE.

Lamport, L. 1977. Concurrent reading and writing. *Commun. ACM 20*, 11 (November), 806–811.

Lamport, L. 1986. On interprocess communication, parts I and II. *Distributed Computing 1*, 77–101.

Lehoczky, J., Sha, L., and Ding, Y. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Tenth IEEE Symposium on Real-Time Systems* (December 1989), pp. 166–171. IEEE.

Leung, J. Y. T. and Whitehead, J. 1982. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation 2*, 4, 237–250.

Liu, C. and Layland, J. 1973. Scheduling algorithms for multiprogramming in a hard real–time environment. *Journal of the ACM 30*, 46–61.

Massalin, H. 1992. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* Ph. D. thesis, Columbia University, New York, New York.

Peterson, G. L. 1983. Concurrent reading while writing. *ACM Trans. on Programm. Lang. Syst. 5*, 1, 46–55.

Rajkumar, R. 1991. *Synchronization In Real-Time Systems – A Priority Inheritance Approach.* Kluwer Academic Publishers, Boston.

Ramamurthy, S., Moir, M., and Anderson, J. H. 1996. Real-time object sharing with minimal support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (May 1996), pp. 233–242. ACM.

Sha, L., Rajkumar, R., and Lehoczky, J. 1990. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers 39*, 9, 1175–1185.

Sorensen, P. 1974. *A Methodology for Real-Time System Development.* Ph. D. thesis, University of Toronto, Toronto, Canada.

Sorensen, P. and Hemachar, V. 1975. A real-time system design methodology. *INFOR 13*, 1, 1–18.

Stone, D. L. 1995. *Managing the Effect of Delay Jitter on the Display of Live Continuous Media.* Ph. D. thesis, University of North Carolina, Chapel Hill, North Carolina.