Real-Time Computing with Lock-Free Shared Objects*

James H. Anderson Srikanth Ramamurthy Kevin Jeffay
Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175

Abstract

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 not locked. 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 object-sharing approaches can be expected to incur much less overhead than approaches based on wait-free objects 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.

1 Introduction

Lock-based approaches to synchronization are the accepted means for interprocess communication in real-time systems. The main problem that arises in such approaches is that of priority inversion, i.e., the situation in which a given task waits on another task of lower priority to unlock a semaphore. Mechanisms such as the *priority ceiling protocol* (PCP) [17, 18] are used to solve this problem. The PCP requires the operating system to identify those tasks that may lock a semaphore. This information is used to ensure that the priority of a task holding a semaphore is at least that of the highest-priority task that ever locks that semaphore. Although the PCP provides a general framework for real-time synchronization, this generality comes at a price, specifically operating system overhead that is sometimes excessive.

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 [3, 7, 16] — 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 [15]. We then compare lock-free and lock-based approaches, both formally, based on our scheduling conditions, and experimentally, based on work involving a real-time desktop videoconferencing facility.

Our formal analysis and experimental work both lead to the same conclusion: lock-free objects often require less overhead than conventional lock-based object-sharing approaches. 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 in the PCP that record the highest-priority task that locks each semaphore). Furthermore, with lock-based schemes, when a high-priority task tries to access an object that is locked by a low-priority task, two "additional" context switches are required, one from the high-priority task to the low-priority task, and another from the low-priority task back to the high-priority

*The first two authors were supported by NSF contract CCR 9216421, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAHO4-95-1-0323. The third author was supported by grants from Intel and IBM.
Email: {anderson, ramamurt, jeffay}@cs.unc.edu.

task. Such context switching is unnecessary when lock-free objects are used.

Lock-free operations are usually implemented using "retry loops" [3, 6, 7, 11, 16]. Figure 1 depicts an example of such an operation, an enqueue taken from a shared queue implementation given in [16]. In this example, an enqueue is performed by trying to thread an item onto the tail of the queue by using a two-word compare-and-swap (CAS2) instruction.[1] This threading is attempted repeatedly until it succeeds. (Note that the queue is not actually "locked" by any task.) A related notion to that of a lock-free object, which may be familiar to some readers, is that of a wait-free object. *Wait-free* objects guarantee a strong form of lock-freedom that precludes all waiting dependencies among tasks [6, 7, 19] (including potentially unbounded retry loops).[2] 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 usually superior for real-time computing on uniprocessors.

From a real-time perspective, lock-free objects are of interest because they do not give rise to priority inversions, and can be implemented with minimal operating system support. Despite these advantages, it may seem that unbounded retry loops render such objects useless in hard real-time systems. Nonetheless, we show that if tasks on a uniprocessor are scheduled appropriately, then such 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 a successful one.

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 an uniprocessor, this can only happen if T_i preempts the access of T_j and then updates B successfully. However, T_i preempts T_j only if T_i has higher priority than T_j. Thus, at each priority level, there is a correlation between failed updates and task preemptions. The maximum number of task preemptions within a time interval can be determined from the timing requirements of the tasks. Using this information, it is possible to determine a bound on the number of failed updates in that interval. 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.

type *objtype* =**record** *data*: *valtype*; *next*: $*$*objtype* **end**
shared var *queue_tail*: $*$*objtype*
procedure *Enqueue*(*input*: *valtype*)
local var *old_tail, new_tail*: $*$*objtype*
begin
  $*$*new_tail* := (*input, NULL*);
  repeat *old_tail* := *queue_tail*
  until CAS2(*queue_tail, queue_tail* −> *next,*
        old_tail, NULL,
        new_tail, new_tail)
end

Figure 1: Lock-free enqueue operation.

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. An important question, then, is how costly lock-free retry loops are likely to be. Although such loops could be long in principle, as shown in [16], 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 (see [2] for a detailed discussion of this issue). The overriding conclusion to be drawn from our work is that, for all but certain pathological objects that require costly retry loops, lock-free objects are likely to require substantially less overhead than lock-based objects implemented using the PCP or other approaches.

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 [20, 21]. Sorenson and Hamacher's work involved a real-time communication mechanism based on wait-free read/write buffers. In their approach, buffers are managed by 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 work presented by Kopetz and Reisinger in [12] and by Johnson and Harathi in [11]. In the former paper, a simple lock-free, one-writer, read/write buffer is presented, and scheduling condi-

[1] The first two parameters of CAS2 are shared variables, the next two parameters are values to which the shared variables are compared, and the last two parameters are new values to be placed in the shared variables should both comparisons succeed. Note that it is possible to simulate the CAS2 instruction in software, as discussed in Section 7.

[2] 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.

tions 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, notation, and two key lemmas. We then derive RM and EDF scheduling conditions in Sections 3 and 4, respectively. In addition, we briefly consider Deadline-Monotonic (DM) scheduling in Section 3. We then formally and experimentally compare lock-free object sharing with other approaches in Sections 5 and 6, respectively. We conclude in Section 7.

2 Definitions and Notation

We use the term *task* to refer to a sequential program that is invoked repeatedly. We call a single execution of a task a *job*. The time at which a job arrives for execution is called its *release time*. A task is *periodic* iff the interval between job arrivals is constant. In our analysis, we assume that all tasks are periodic and share a single processor. We assume that all release times and periods are integers. 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.

As in the previous section, we call an iteration of a lock-free loop a *successful update* if it results in the successful completion of the corresponding operation, and a *failed update* otherwise. For now, we assume that the *deadline* of a job is the end of the corresponding period. Later, when considering DM scheduling at the end of Section 3, we relax this assumption. A task set is *schedulable* iff all tasks meet their deadlines at all times. 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 $r_i(k) = r_i(0) + k \cdot p_i$. We use k as a job index. Unless stated otherwise, we assume that k is universally quantified with range $k \geq 0$.
- 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_m - The m^{th} shared object in the system.
- 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. This is also the extra computation required in the event of a failed update.

We obtain conditions for schedulability by determining the worst-case "unfulfilled demand" of each task. Informally, the unfulfilled demand of task T_i at time t is the remaining computation time of T_i's current job. In the derivation of our scheduling conditions, we assume that the unfulfilled demand of T_i increases by s — the computation time of the extra loop iteration that will result from a failed update — when a job of T_i is preempted by a higher-priority job that accesses a common lock-free object. This approach is pessimistic because the preempted job may not, in fact, be accessing any shared object when preempted. Before we present our scheduling conditions, we define the concept of a "busy point" and then state two lemmas used in the proofs of these conditions.

In [15], it is shown that for independent tasks, 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 synchronize using lock-free objects (see [2] for examples showing why this is not so). Instead of defining the critical instant or giving the worst-case phasing of the tasks, we introduce the notion of a *busy point*. The busy point of the k^{th} job of task T_i is denoted by $b_i(k)$, where $k \geq 0$. 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 zero unfulfilled demand.

It is easy to show that $b_i(k)$ is well-defined for any i and k. In particular, at time 0, each task has either just released its first job or has no unfulfilled demand. Hence, $0 \leq b_i(k) \leq r_i(k)$. (For the RM scheme, it is possible to prove a tighter bound of $(r_i(k-1), r_i(k)]$ on the range of $b_i(k)$. See [2] for details.) In the formal proofs of our scheduling conditions, we inductively count the number of failed updates over intervals of time. A busy point provides a convenient instant at which to start such an inductive argument, because tasks that have zero unfulfilled demand or that have just released a job have no failed updates.

We now state two lemmas that bound the number of failed updates in a given interval, under the RM and EDF schemes. Full proofs of these lemmas can be found in [2].

Lemma 2.1: *Consider the k^{th} job of task T_i and any $t \in [b_i(k), r_i(k+1))$, where $k \geq 0$. (i) Under the RM scheme, the number of failed updates in T_i and higher-priority tasks in the interval $[b_i(k), t]$ is at most $\sum_{j=1}^{i-1} \left\lceil \frac{t-b_i(k)}{p_j} \right\rceil$. (ii) Under the EDF scheme, the number of failed updates in jobs with a deadline at or before t in the interval $[b_i(k), t]$ is at most $\sum_{j=1}^{N} \frac{t-b_i(k)}{p_j}$.* □

The above lemma states that the number of failed updates in an interval $[b_i(k), t]$ is at most the number of higher-priority jobs released in the interval $[b_i(k)+1, t]$.

Lemma 2.2: *Under the RM scheme, If the k^{th} job of task T_i has not completed execution at some time $t' \in (r_i(k), r_i(k+1)]$, where $k \geq 0$, then, for any t in the interval $[b_i(k), t')$, 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.* □

3 RM/DM Conditions

The following 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 failed updates. The second summation represents the total additional demand placed on the processor due to failed updates in T_i and higher-priority tasks. The right-hand side of the expression is the available processor time in $[0, t)$. As noted in the introduction, this condition can be applied without knowledge of which tasks access which objects.

Theorem 3.1: *A set of tasks scheduled under the RM scheme is schedulable 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 the k^{th} job of some task T_i be the first job to miss its deadline. This can only happen if T_i has positive unfulfilled demand at $r_i(k+1)-1$. Consider any t in the interval $[b_i(k), r_i(k+1))$. By Lemma 2.2, 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.

We now derive a bound on $D_i(b_i(k), t)$, the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$. $D_i(b_i(k), t)$ is comprised of the demand placed by job releases and the extra demand placed by failed updates. Recall that at the busy point of the k^{th} job, T_i and all higher-priority task have either completed execution (have no unfulfilled demand) or have a job release. Each job of some task T_j can place a demand of c_j, 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 2.1, the number of failed updates in the interval $[b_i(k), t]$ is given by $\sum_{j=1}^{i-1} \lceil (t - b_i(k))/p_j \rceil$. Each failed update requires s units of additional demand. Therefore, the total additional demand due to failed updates is at most $\sum_{j=1}^{i-1} \lceil (t - b_i(k))/p_j \rceil s$. Therefore, we have $D_i(b_i(k), t) \leq \sum_{j=1}^{i} \left\lceil \frac{t-b_i(k)+1}{p_j} \right\rceil c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-b_i(k)}{p_j} \right\rceil s$.

As stated previously, 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.

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

Using the bound on $D_i(b_i(k), t)$, the previous expression can be rewritten as follows.

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

The above expression holds for all t in the interval $[b_i(k), r_i(k+1))$. Because the above 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, from the definitions, $r_i(k+1) - r_i(k) = p_i$. Therefore, the previous expression holds for all t in $(0, p_i]$. □

In the videoconferencing system described in Section 6, job deadlines and release points for the given task set do not necessarily coincide, as we have assumed. However, the scheduling condition of Theorem 3.1 can easily be adapted to apply to such a task set. 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 scheduling condition. This condition assumes that priority is assigned by the DM scheme [14], in which tasks with smaller relative deadlines have higher priorities. The two summation terms in the stated expression below give the computational demand of T_i and higher-priority tasks, and the additional computation required due to failed updates, respectively, in an interval of length t.

Theorem 3.2: *A set of tasks scheduled under the DM scheme is schedulable if the following condition holds for every task T_i.*

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

In comparing this condition to that given in Theorem 3.1, 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 RM scheduling.

4 EDF Scheduling Condition

The following theorem gives a sufficiency condition for schedulability under the EDF scheme. Like the RM sufficiency condition of the previous section, this condition can be applied without knowledge of which tasks access which objects.

Theorem 4.1: *A set of periodic tasks scheduled under the EDF scheme is schedulable 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 $\sum_{j=1}^{N} \frac{c_j+s}{p_j} > 1$. Assume that the given task set is not schedulable. Let the k^{th} job of some task T_i be the first job to miss its deadline. This can only happen 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.

We first derive a bound on $D_i(b_i(k), r_i(k+1)-1)$, the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), r_i(k+1))$. $D_i(b_i(k), r_i(k+1)-1)$ is comprised of the demand placed by job releases and the extra demand placed by failed updates. Recall that at the busy point of the k^{th} job, all jobs of equal or higher priority have either completed execution (have no unfulfilled demand) or have a job release. Each job of some task T_j can place a demand of c_j, and there are at most $(r_i(k+1) - b_i(k))/p_j$ job releases of that task in the interval $[b_i(k), t]$ 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 $\sum_{j=1}^{N} \frac{(r_i(k+1)-b_i(k)) \cdot c_j}{p_j}$. By Lemma 2.1, The total number of failed updates in the interval $[b_i(k), r_i(k+1))$ is bounded by the term $\sum_{j=1}^{N} \frac{r_i(k+1)-b_i(k)-1}{p_j}$. Each failed update requires s units of additional demand. Therefore, the total additional demand due to failed updates is at most $\sum_{j=1}^{N} \frac{(r_i(k+1)-b_i(k)-1) \cdot s}{p_j}$. As stated previously, the difference between the total demand placed on the processor by jobs with deadlines 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.

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

The terms on left-hand side of the previous expression give the total demand placed by jobs with deadlines before $r_i(k+1)$, the total additional demand due to failed updates in those jobs, and the available processor time, in the interval $[b_i(k), r_i(k+1))$, respectively. The above expression can be rewritten as follows.

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

The previous expression 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 equation, we have the following expression, thus completing our proof.

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

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 computation time per object access, i.e., s for lock-free objects and r for lock-based objects. As explained below, recent studies that evaluate the performance of lock-free objects [16] and lock-based objects [5] indicate that s is likely to be much smaller than r. 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 synchronization under RM scheduling with the overhead of the lock-based priority ceiling protocol (PCP) [17]. 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 [17], is defined as follows.

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

Observe that $\langle \forall j : j \leq i : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \ \wedge$ *sched_PCP* implies $\langle \forall i\ \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot s) + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle$. Because $c_j = u_j + m_j \cdot s$, the previous expression is equivalent to the scheduling condition of Theorem 3.1. 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.

What are typical values of s and r? A performance comparison of various lock-free objects is given by Massalin in [16]. 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 [5]. 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 31 microseconds, while r is 126.5 microseconds.

In the above comparison, we have actually ignored the effect of blocking under the PCP. If the blocking times are considerable, then lock-free objects would perform better than as indicated above. 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 a failed update as we have assumed.

5.2 Dynamic-Priority Scheduling

We now compare the overhead of lock-free objects with two dynamic-priority schemes that use semaphore-based objects: the dynamic priority ceiling protocol (DPCP) [4], and the dynamic deadline modification (DDM) scheme under EDF scheduling (EDF/DDM) [8]. Based on the analysis in [4], a sufficient condition for the schedulability of a set of periodic tasks under the DPCP, *sched_DPCP*, can be defined as follows.

$$sched\_DPCP \equiv \sum_{j=1}^{N} \frac{(c_j + block_j)}{p_j} \leq 1$$

In the above condition, $block_j$ is the maximum time for which task T_j can be blocked by some lower-priority task, and equals the time to execute the longest critical section. Since we have assumed that semaphore-based accesses require at most r time units, we have $block_j = r$.

Observe that the above condition resembles the one we have derived for lock-free objects. It can be easily shown that if $(s < r \wedge sched\_DPCP)$, then $\sum_{j=1}^{N}(c_j + s)/p_j \leq 1$. Therefore, by Theorem 4.1, if $s < r$ then any set of tasks that can be scheduled under the DPCP can also be scheduled using lock-free objects. Because s is likely to be smaller than r, processor utilization is likely to be smaller when lock-free objects are used for synchronization. Thus, there are task sets that can be scheduled when lock-free objects are used but not when DPCP is used.

We now turn our attention to the EDF/DDM scheme presented in [8]. 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. Based on the analysis of [8], a sufficient condition for the schedulability of a set of periodic tasks under the EDF/DDM scheme, *sched_DDM*, can be defined as follows.

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

Observe that $\langle \forall j : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \wedge$ *sched_DDM* implies $\sum_{j=1}^{N} \frac{u_j + (m_j+1) \cdot s}{p_j} \leq 1$. Because $c_j = u_j + m_j \cdot s$, the pervious expression is equivalent to the scheduling condition of Theorem 4.1. 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 much smaller than r.

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 algorithms ensure termination by requiring each task to "help" every other task to complete any pending object access [6, 7]. However, 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 Section 6.2.

6 Experimental Comparison

In this section, we provide empirical evidence that lock-free objects are always competitive with, and 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 [10]. 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 virtually 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 lock-free shared objects are usually superior to wait-free objects 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. We modified the portion of the system responsible for the acquisition, compression, and network transmission of media samples by the sending workstation.

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. Queues of shared media samples exist between the digitizing task and the compression task and between the compression task and the network transmission task. The real-time constraints on the operation of the pipeline require media samples to flow through the pipeline in a periodic manner. Each stage of the pipeline must process a media sample every 33 milliseconds, and no media samples may be lost due to buffer overflows. These constraints are met by implementing the pipeline as a set of periodic tasks.

A comprehensive view of the tasks and shared queues on the sending workstation is given in Figure 1. In this figure, an arrow is directed from each task to each of the shared objects it accesses. The implicit resources ($S_1 - S_{13}$) correspond to queues used for inter-task communication. These queues do not contain any media samples. For our purposes, it suffices to consider the tasks in Figure 1 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 [22].

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 by Massalin in [16], and wait-free queues by using the wait-free universal construction given by Herlihy in [7]. 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.

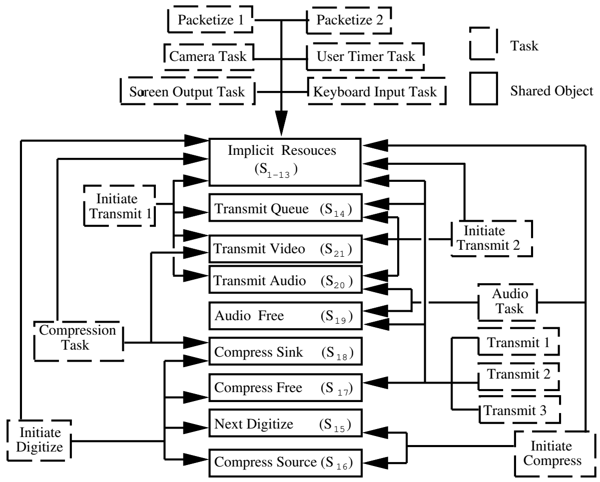

Figure 1: Tasks and shared queues in the videoconferencing system.

We found that the videoconferencing task set was not schedulable under the DM scheduling when the shared queues were implemented using Herlihy's wait-free universal construction. This is due to the overhead of helping, as discussed in Section 5.3. In contrast, our lock-free implementations required very little overhead, with failed updates occurring only rarely. For example, in ten executions of the system, only 363 failed updates occurred in 415,229 enqueue operations. We also found that multiple failed updates by 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 a DM scheduling algorithm [14].

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, the details of which can be found in [2]. The analysis shows that under the PCP the task `Packetize 2` is not schedulable. This task copies compressed media sample buffers to the network adapter. When `Packetize 2` 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. The analysis also predicts 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. (In our system, s equals 31 microseconds and r equals 126.5 microseconds. Observe that s is less than $r/2$.)

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 set of tasks in the system is schedulable under both schemes. This result is predicted by the formal analysis of the system. For brevity, the formal analysis is not presented here (refer to [2] for details).

In order to more precisely compare lock-free objects with objects implemented under the EDF/DDM scheme, we introduced a dummy task 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 should 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 the 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 are easier to use, because their application does not require detailed knowledge of which tasks access which objects. Second, systems using lock-free objects can be more easily modified to add tasks dynamically, since operating system tables do not have to be recomputed. Third, in contrast to the PCP, lock-free accesses do not give rise to excessive context switches. Finally, and most importantly, lock-free objects usually entail substantially less overhead than objects implemented using lock-based techniques. This is the case for most common objects, the exception being certain pathological objects that require excessive copying and hence costly retry loops.

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. In the case of the DPCP, if we introduce the blocking factor into our sufficient condition, then we require $s + y$ to be at most r for lock-free objects to perform as well as lock-based objects under the DPCP. Note that, since y is smaller than s, we have $s \leq r/2 \Rightarrow s + y \leq r$.

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. Recently, Anderson and Moir presented algorithms for implementing multi-object operations that allow similar functionality in lock-free (and wait-free) implementations [1]. Using these algorithms, a buffer transfer can be accomplished in a lock-free (or wait-free) manner by simultaneously accessing 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, presumably a loop that accesses several objects at once.

Acknowledgements: 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 help-

ing with the experimental work described in Section 6.

References

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

[2] J. Anderson, S. Ramamurthy and K. Jeffay, "Real-Time Computing Using Lock-free Shared Objects", Technical Report TR95-021, Department of Computer Science, University of North Carolina, June 1995 (`URL: http://www.cs.unc.edu/~anderson/papers/rtss95.ps.Z`).

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

[4] M. I. Chen and K. J. Lin, "Dynamic Priority Ceiling: A Concurrency Control Protocol for Real Time Systems", *Real-Time Systems Journal*, Vol. 2, No. 1, 1990, pp. 325-346.

[5] B. O. Gallmeister and C. Lanier, "Early Experience With POSIX 1003.4 and POSIX 1003.4A", *Proceedings of the 12th IEEE Real-Time Systems Symposium*, 1991, pp. 190-198.

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

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

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

[9] K. Jeffay and D. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems", *Proceedings of the 14^{th} IEEE Symposium on Real-Time Systems*, Durham, NC, 1993, pp. 212-221.

[10] K. Jeffay, D.L. Stone, and F.D. Smith, "Kernel Support for Live Digital Audio and Video", *Computer Communications*, Vol. 15, No. 6, July/August 1992, pp. 388-395.

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

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

[13] 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.

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

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

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

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

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

[19] A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register", *Journal of the ACM*, Vol. 41, No. 2, March 1994, pp. 311-339.

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

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

[22] D. Stone, *Managing the Effect of Delay Jitter on the Display of Live Continuous Media*, Doctoral Dissertation, University of North Carolina, Chapel Hill, 1995.