Efficient Object Sharing in Quantum-Based Real-Time Systems*

James H. Anderson, Rohit Jain, and Kevin Jeffay
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{anderson,jain,jeffay}@cs.unc.edu

Abstract

We consider the problem of implementing shared objects in uniprocessor and multiprocessor real-time systems in which tasks are executed using a scheduling quantum. In most quantum-based systems, the size of the quantum is quite large in comparison to the length of an object call. As a result, most object calls can be expected to execute without preemption. A good object-sharing scheme should optimize for this expected case, while achieving low overhead when preemptions do occur. In this paper, we present several new shared-object algorithms for uniprocessors and multiprocessors that were designed based upon this principle. We also present scheduling analysis results that can be used in conjunction with these algorithms.

1. Introduction

In many real-time systems, tasks are scheduled for execution using a scheduling quantum. Under quantum-based scheduling, processor time is allocated to tasks in discrete time units called *quanta*. When a processor is allocated to some task, that task is guaranteed to execute without preemption for Q time units, where Q is the length of the quantum, or until it terminates, whichever comes first. Many real-time applications are designed based on scheduling disciplines such as proportional-share [20] and round-robin scheduling that are expressly quantum-based. Under proportional-share scheduling, each task is assigned a *share* of the processor, which represents the fraction of processing time that that task should receive. Quanta are allocated in a manner that ensures that the amount of processor time each task receives is commensurate with its share. Round-robin scheduling is a simpler scheme in which each task has an identical share.

Quantum-based execution also arises when conventional priority-based scheduling disciplines, such as rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling, are implemented on top of a timer-driven real-time kernel [14]. In such an implementation, interrupts are scheduled to occur at regular intervals, and scheduling decisions are made when these interrupts occur. The length of time between interrupts defines the scheduling quantum. Timer-driven systems can be seen as a compromise between nonpreemptive and completely preemptive systems. In fact, nonpreemptive and preemptive systems abstractly can be viewed as the extreme endpoints in a continuum of quantum-based systems: a nonpreemptive system results when $Q = \infty$ and a fully preemptive system results when $Q = 0$. Nonpreemptive systems have several advantages over preemptive systems, including lower scheduling overheads (if preemptions are frequent) and simpler object-sharing protocols [8, 13]. Also, timing analysis is simplified because cache behavior is easier to predict. However, these advantages come at the potential expense of longer response times for higher-priority tasks. Quantum-based systems can be seen as a compromise between these two extremes.

In this paper, we consider the problem of efficiently implementing shared objects in quantum-based real-time systems. We consider both uniprocessor and multiprocessor systems. The basis for our results is the observation that, in most quantum-based systems, the size of the quantum is quite large compared to the length of an object call. Indeed, processors are becoming ever faster, decreasing object-access times, while quantum sizes are not changing. Even with the technology of several years ago, one could make the case that object calls are typically short compared to a quantum. As evidence of this, we cite results from experiments conducted by Ramamurthy to compute access times for several common objects [18]. These experiments were performed on a 25 MHz 68030 machine and involved objects ranging from queues to linked lists to medium-sized

*Work supported by NSF grant CCR 9510156. The first author was also supported by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323, by NSR grant CCR 9732916, and by an Alfred P. Sloan Research Fellowship. The second author was supported by a UNC Board of Governor's Fellowship. The third author was supported by a grant from IBM Corporation.

balanced trees. Both lock-based and lock-free (see below) object implementations were evaluated. Ramamurthy found that, even on a slow 25 MHz machine, all object calls completed within about 100 microseconds, with most taking much less. In contrast, a quantum in the range 1-100 milliseconds is used in most quantum-based systems.[1]

These numbers suggest that, in a quantum-based system, most object calls are likely to execute without preemption. A good object-sharing scheme should optimize for this expected case, while achieving low overhead when preemptions do occur. Clearly, an optimistic object-sharing scheme is called for here, because pessimistically defending against interferences on every object call by acquiring a lock will lead to wasted overhead most of time. In an *optimistic scheme*, objects are accessed in a manner that does not preclude interferences due to concurrent accesses. If an operation on an object *is* interfered with before it is completed, then it has no effect on the object. Any operation that is interfered with must be retried in order to complete. Optimistic schemes perform well when retries are rare, which is precisely the situation in quantum-based systems.

In this paper, we show that it is possible to significantly optimize retry-based shared-object algorithms by directly exploiting the relative infrequency of preemptions in quantum-based systems. The specific assumption we make throughout this paper regarding preemptions is as follows.[2]

Preemption Axiom: *The quantum is large enough to ensure that each task can be preempted at most once across two consecutive object calls.* □

Given the Preemption Axiom, each object call can be retried at most once, i.e., there is a bound on overall object-sharing costs. The Preemption Axiom is quite liberal: not only are object calls of short to medium duration allowed (the case we most expect and optimize for), but also calls that are quite long, approaching the length of an entire quantum.

Our work builds upon recent research by us and others on using lock-free and wait-free shared-object algorithms in real-time systems [3, 4, 5, 6, 15, 19]. Operations on lock-free objects are optimistically performed using a user-level retry loop. Such an operation is atomically validated and committed by invoking a synchronization primitive such as *compare-and-swap* (`CAS`). The retry loop is executed repeatedly until this validation step succeeds. Wait-free shared objects are required to satisfy an extreme form of lock-freedom that precludes all waiting dependencies among tasks, including potentially unbounded operation retries.

The remainder of this paper is organized as follows. In the first part of the paper (Section 2), we consider the problem of implementing shared objects in quantum-based uniprocessor systems. Our approach is to develop lock-free algorithms that are optimized in accordance with the Preemption Axiom. The Preemption Axiom ensures that each lock-free operation is retried at most once. Thus, if an operation is interfered with due to a preemption, then the retry code can be purely sequential code in which shared data is read and written without using synchronization primitives. In short, the Preemption Axiom automatically converts a lock-free implementation into a wait-free one. In addition to discussing algorithmic techniques, we also show how to account for object-sharing costs in scheduling analysis.

In the second part of the paper (Section 3), we consider the problem of implementing shared objects in quantum-based multiprocessor systems. In a multiprocessor, a retry mechanism by itself clearly is not sufficient, because a task on one processor may be repeatedly interfered with due to object invocations by tasks on other processors. Our approach is to use a retry mechanism in conjunction with a preemptable queue lock [17]. In our approach, a task performs an operation on an object by first acquiring a lock; if a task is preempted before its operation is completed, then its operation is retried. In comparison to previous preemptable queue-lock algorithms [21, 22], ours is quite simple. Its simplicity is mostly due to the fact that it was designed for systems satisfying the Preemption Axiom.

2. Uniprocessor Systems

In this section, we consider the implementation of shared objects in quantum-based uniprocessor systems. We also show how to account for object-sharing costs arising from the proposed implementations in scheduling analysis.

2.1. Implementing Objects

The Preemption Axiom ensures that each lock-free operation is retried at most once. Thus, if an operation is interfered with due to a preemption, then the retry code can be optimized to be purely sequential code in which shared data is accessed without using synchronization primitives.

Implementing read-modify-writes. As an example of an implementation that is optimized in this way, consider Figure 1. This figure shows how to implement read-modify-write (RMW) operations using `CAS`.[3] A RMW operation on a variable X is characterized by specifying a function f.

[1] A quantum of 1-100 milliseconds may not be sufficient for all real-time applications. For systems that employ a very small quantum, the results of this paper may not be applicable.

[2] This axiom can be weakened to allow preemptions by tasks due to external interrupts, provided that each task can be preempted at most once across two consecutive object calls by other tasks that access shared objects, and the time spent servicing external interrupts is accounted for when analyzing schedulability.

[3] `CAS`(*addr*, *old*, *new*) is equivalent to the atomic code fragment $\langle$ **if** $*addr = old$ **then** $*addr := new$; **return** *true* **else return** *false* **fi** $\rangle$.

```
procedure RMW(Addr: ptr to valtype; f: function) returns valtype
private variable old, new: valtype
1:  old := *Addr;
2:  new := f(old);
3:  if CAS(Addr, old, new) = false then
4:     old := *Addr;                              /* retry operation */
5:     *Addr := f(old)  /* lines 4-5 execute without preemption */
    fi;
6:  return old
```

Figure 1. Uniprocessor read-modify-write implementation.

Informally, such an operation has the effect of the following atomic code fragment: $\langle x := X;\ X := f(x);$ **return** $x\rangle$. Example RMW operations include fetch-and-increment, fetch-and-store, and test-and-set.

The implementation in Figure 1 is quite simple. If the CAS at line 3 succeeds, then the RMW operation atomically takes effect when the CAS is performed. If the CAS fails, then the invoking task must have been preempted between lines 1 and 3. In this case, the Preemption Axiom implies that lines 4 and 5 execute without preemption. Given this implementation, we can conclude that, in any quantum-based uniprocessor system that provides CAS, any object accessed only by means of reads, writes, and read-modify-writes can be implemented in constant time. It should be noted that virtually every modern processor either provides CAS or instructions that can be used to easily implement CAS.

Conditional compare-and-swap. Using similar principles, it is possible to efficiently implement *conditional compare-and-swap* (CCAS), which is a very useful primitive when implementing lock-free and wait-free objects. CCAS has the following semantics.

```
CCAS(V: ptr to vertype; ver: vertype; X: ptr to wdtype;
                              old, new: wdtype) returns boolean
  ⟨ if *V ≠ ver ∨ *X ≠ old then return false fi;
    *X := new;
    return true ⟩
```

The angle brackets above indicate that CCAS is atomic. As its definition shows, CCAS is a restriction of a two-word CAS primitive in which one word is a compare-only value. Lock-free and wait-free objects can be implemented by using a "version number" that is incremented by each object call [3, 12]. CCAS is useful because the version number can be used to ensure that a "late" CCAS operation performed by a task after having been preempted has no effect.

Figure 2 shows how to implement CCAS using CAS on a quantum-based uniprocessor. The implementation works by packing a task index into the words being accessed. The

```
type wdtype = record val: valtype; task: 0..N end
     /* all fields of wdtype are stored in one word; task indices ... */
        /* ... range over 1..N; the task field should be 0 initially */

procedure CCAS(V: ptr to vertype; ver: vertype;
     W: ptr to wdtype; old, new: wdtype; p: 1..N) returns boolean
private variable w: wdtype
           /* p is assumed to be the identify of the invoking task */

1:  w := *W;
2:  if w.val ≠ old.val then return false fi;
3:  if *V ≠ ver then return false fi;
4:  if CAS(W, w, (old.val, p)) then
5:     if *V ≠ ver then
6:        w := *W;     /* lines 6-8 execute without preemption */
7:        *W := (w.val, 0);
8:        return false
       fi;
9:     if CAS(W, (old.val, p), (new.val, 0)) then return true fi
    fi;
    /* lines 10-13 execute without preemption */
10: if W->val ≠ old.val then return false fi;
11: if *V ≠ ver then return false fi;
12: *W := (new.val, 0);
13: return true

procedure Read(W: ptr to wdtype) returns wdtype
private variable w: wdtype

14: w := *W;
15: if w.task = 0 then return w.val
    else
16:    CAS(W, w, (w.val, 0)); /* lines 16-19 are rarely executed */
17:    w := *W;
18:    CAS(W, w, (w.val, 0));
19:    return w.val
    fi
```

Figure 2. CCAS implementation. Code for reading a word accessed by CCAS is also shown.

task index field is used to detect preemptions. It is clearly in accordance with the semantics of CCAS for a task T_i to return from line 2 or 3. To see that the rest of the algorithm is correct, observe that a task T_i can find $*V \neq ver$ at line 5 only if it was preempted between lines 3 and 5. Similarly, the CAS operations at lines 4 and 9 can fail only if a preemption occurs. By the Preemption Axiom, this implies that lines 6-8 and 10-13 execute without preemption. It is thus easy to see that these lines are correct. The remaining possibility is that a task T_i returns from line 9. T_i can return here only if the CAS operations performed by T_i at lines 4 and 9 both succeed. The first of these CAS operations only updates the task index field of W; the second updates the value field. We claim that T_i's CAS at line 9 is successful only if no task performs a Read operation on word W or assigns W within its CCAS procedure between the execution of lines 4 and 9 by T_i — note that this property implies that T_i's CCAS can be linearized to its execution of line 5. To see that this property holds, observe that if some other task updates W in its CCAS procedure, then $W{-}{>}task \neq i$ is established, implying that T_i's CAS at line 9 fails. Also, if some task T_j performs a Read operation on W when $W{-}{>}task = i$ holds, then it must establish $W{-}{>}task = 0$, causing T_i's CAS at line 9 to fail. To see this, note that, by the Preemption Axiom, T_j's execution of the Read procedure itself can be preempted at most once. By inspecting the code of this procedure, it can be seen that this implies that T_j must establish $W{-}{>}task = 0$ during the same quantum as when it reads the value of W.

It is important to stress that our objective here is to design object implementations that perform very well in the absence of preemptions and that are still correct when preemptions do occur. If the code in Figure 2 is never preempted when executed by any task, then lines 6-8, 10-13, and 16-19 are never executed. Thus, in the expected case, this object implementation should perform well.

In the full paper [1], an implementation of a multi-word CAS (MWCAS) object is presented that is based on techniques that are similar to those described above; this implementation is not included here due to lack of space. The semantics of MWCAS generalizes that of CAS to allow multiple words to be accessed simultaneously. MWCAS is a useful primitive for two reasons. First, it simplifies the implementation of many lock-free objects; queues, for instance, are easy to implement with MWCAS, but harder to implement with single-word primitives. Second, it can be used to implement multi-object operations. For example, an operation that dequeues an item off of one queue and enqueues it onto another could be implemented by using MWCAS to update both queues. Our MWCAS implementation is a bit more involved than those described above, and thus may be of interest to readers interested in techniques for implementing more complicated objects.

2.2. Scheduling Analysis

We now turn our attention to the issue of accounting for object-sharing costs in scheduling analysis when object implementations like those proposed in the previous subsection are used. We consider scheduling analysis under the rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling schemes. We also very briefly consider proportional-share (PS) scheduling.

We begin by considering the RM and EDF schemes. In both of these schemes, a periodic task model is assumed. We call each task invocation a *job*. For brevity, we limit our attention to systems in which each task's relative deadline equals its period (extending our results to deal with systems in which a task's relative deadline may be less than its period is fairly straightforward). In our analysis, we assume that each job is composed of distinct nonoverlapping computational fragments or *phases*. Each phase is either a *computation phase* or an *object-access phase*. Shared objects are not accessed during a computation phase. An object-access phase consists of exactly one retry loop. We assume that tasks are indexed such that, if a job of task T_i can preempt a job of task T_j, then $i < j$ (such an indexing is possible under both RM and EDF scheduling). The following is a list of symbols that will be used in our analysis.

- N - The number of tasks in the system. We use i, j, and l as task indices; each is universally quantified over $\{1, \ldots, N\}$.
- Q - The length of the scheduling quantum.
- p_i - The period of task T_i.
- w_i - The number of phases in a job of task T_i. The phases are numbered from 1 to w_i. We use u and v to denote phases.
- x_i - The number of object-access phases in a job of task T_i.
- c_i^v - The worst-case computational cost of the v^{th} phase of task T_i, where $1 \leq v \leq w_i$, assuming no contention for the processor or shared objects. We denote total cost over all phases by $c_i = \sum_{v=1}^{w_i} c_i^v$.
- r_i^v - The cost of a retry if the v^{th} phase of task T_i is interfered with. For computation phases, $r_i^v = 0$. For object-access phases, we usually have $r_i^v < c_i^v$, because retries are performed sequentially. We let $r_i = max_v(r_i^v)$.
- $m_i^v(j, t)$ - The worst-case number of interferences in T_i's v^{th} phase due to T_j in an interval of length t.
- f_i^v - An upper bound on the number of interferences of the retry loop in the v^{th} phase of T_i during a single execution of that phase.

A simple bound on interference costs. The simplest way to account for object interference costs is to simply inflate each task T_i's computation time to account for such costs. This can be done by solving the following recurrence.

$$c_i' = c_i + min[x_i, (\left\lceil \frac{c_i'}{Q} \right\rceil - 1)] \cdot r_i \qquad (1)$$

c_i' is obtained here by inflating c_i by r_i for each quantum boundary that is crossed, up to a maximum of x_i such boundaries (since T_i accesses at most x_i objects in total). If task T_i accesses objects with widely varying retry costs, then the above recurrence may be too pessimistic. Let $r_{i,1}$ be the maximum retry cost of any of T_i's object-access phases, let $r_{i,2}$ be the next-highest cost, and so on. Also, let $v_i = min[x_i, (\lceil c'_i/Q \rceil - 1)]$. Then, we can more accurately inflate c_i by solving the following recurrence.

$$c'_i = c_i + \sum_{k=1}^{v_i} r_{i,k} \tag{2}$$

Once such c'_i values have been calculated, they can be used within scheduling conditions that apply to independent tasks. A condition for the RM scheme is given in the following theorem.

Theorem 1: *In an RM-scheduled quantum-based uniprocessor system, a set of tasks with objects implemented using the proposed retry algorithms is schedulable if the following holds for every task T_i, where $B_i = \min(Q, \max_{j>i}(c'_j))$.*

$(\exists t : 0 < t \leq p_i :: B_i + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil c'_j \leq t)$ $\Box$

In the above expression, B_i is a blocking term that arises due to the use of quantum-based scheduling [14].[4] The next theorem gives a scheduling condition for the EDF scheme.

Theorem 2: *In an EDF-scheduled quantum-based uniprocessor system, a set of tasks with objects implemented using the proposed retry algorithms is schedulable if the following holds.*

$\sum_{i=1}^{N} \frac{c'_i}{p_i} \leq 1 \wedge$
$(\forall i : 1 \leq i \leq N :: (\forall t : p_1 < t < p_i :: min(Q, c'_i)+$
$\sum_{j=1}^{i-1} \lfloor \frac{t-1}{p_j} \rfloor c'_j \leq t))$ $\Box$

The above condition is obtained by adapting the condition given by Jeffay et al. in [13] for nonpreemptive EDF scheduling. Note that this condition reduces to that of Jeffay et al. when $Q = \infty$ and to that for preemptive EDF scheduling [16] when $Q = 0$.

Bounding interference costs using linear programming. Anderson and Ramamurthy showed that when lock-free objects are used in a uniprocessor system, object interference costs due to preemptions can be more accurately bounded using linear programming [4]. Given the Preemption Axiom, we show that it is possible to obtain bounds that are tighter than those of Anderson and Ramamurthy.

Our linear programming conditions make use of a bit of additional notation. If a job of T_j interferes with the v^{th} phase of a job of T_i, then an additional demand is placed on the processor, because another execution of the retry-loop iteration in T_i's v^{th} phase is required. We denote this additional demand by $s_i^v(j)$. Formally, $s_i^v(j)$ is defined as follows.

Definition 1: Let T_i and T_j be two distinct tasks, where T_i has at least v phases. Let z_j denote the set of objects modified by T_j, and a_i^v denote the set of objects accessed in the v^{th} phase of T_i. Then,

$$s_i^v(j) = \begin{cases} r_i^v & \text{if } j < i \wedge a_i^v \cap z_j \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Give the above definition of $s_i^v(j)$, we can state an *exact* expression for the worst-case interference cost in tasks T_1 through T_i in any interval of length t.

Definition 2: The total cost of interferences in jobs of tasks T_1 through T_i in any interval of length t, denoted $E_i(t)$, is defined as follows: $E_i(t) \equiv \sum_{j=1}^{i} \sum_{v=1}^{w_j} \sum_{l=1}^{j-1} m_j^v(l,t) s_j^v(l)$. $\Box$

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

Expression $E_i(t)$ accurately reflects the worst-case additional demand placed on the processor in an interval $\mathcal{I}$ of length t due to interferences in tasks T_1 through T_i. Precisely evaluating this expression is computationally expensive, so we instead will try to obtain a bound on $E_i(t)$ that is as tight as possible. We do this by viewing $E_i(t)$ as an expression to be maximized. The $m_j^v(l,t)$ terms are the "variables" in this expression. These variables are subject to certain constraints. We obtain a bound for $E_i(t)$ by using linear programming to determine a maximum value of $E_i(t)$ subject to these constraints. We now explain how appropriate constraints on the $m_j^v(l,t)$ variables are obtained. In this explanation, we focus on the RM scheme. Defining similar constraints for the EDF scheme is fairly straightforward. We impose six sets of constraints on the $m_i^v(j,t)$ variables.

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

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

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

[4] In [14], it is assumed that timer interrupts are spaced apart by a constant amount of time. If a task completes execution between these interrupts, then the processor is allocated to the next ready task, if such a task exists. This newly-selected task will execute for a length of time that is less than a quantum before possibly being preempted. In our work, we assume that whenever the processor is allocated to a task, that task executes for an entire quantum (or until it terminates) before possibly being preempted. Nonetheless, the blocking calculations due to quantum-based scheduling are the same in both models.

Constraint Set 4: $(\forall i, v :: f_i^v \leq 1)$. □

Constraint Set 5: $(\forall i :: \sum_{j=1}^{i-1} \sum_{v=1}^{w_j} m_i^v(j,t) \leq (\left\lceil \frac{c_i'}{Q} \right\rceil - 1) \cdot \left\lceil \frac{t+1}{p_i} \right\rceil)$. □

Constraint Set 6: $(\forall i :: \sum_{j=1}^{i-1} \sum_{v=1}^{w_j} m_i^v(j,t) \leq x_i \cdot \left\lceil \frac{t+1}{p_i} \right\rceil)$. □

There first three constraint sets were given previously by Anderson and Ramamurthy [4]. The first set of constraints follows because the number of interferences in jobs of T_i due to T_j in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of T_j that can be released in $\mathcal{I}$. The second set of constraints follows from a result presented in [6], which states that the total number of interferences in all jobs of tasks T_1 through T_i in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of tasks T_1 through T_{i-1} released in $\mathcal{I}$. In the third set of constraints, the term f_i^v is an upper bound on the number of interferences of the retry loop in the v^{th} phase of T_i during a single execution of that phase. The reasoning behind this set of constraints is as follows. If at most f_i^v interferences can occur in the v^{th} phase of a job of T_i, and if there are n jobs of T_i released in an interval $\mathcal{I}$, then at most nf_i^v interferences can occur in the v^{th} phase of T_i in $\mathcal{I}$. In Anderson and Ramamurthy's paper, the f_i^v terms are calculated by solving an additional set of linear programming problems. In our case, they can be bounded as shown in the fourth set of constraints.[5] This is because, by the Preemption Axiom, each object access can be interfered with at most once. The last two constraint sets arise for precisely the same reasons as given when recurrence (1) was explained. The c_i' term in the fifth constraint set can be calculated by solving recurrence (1) or (2).

We are now in a position to state scheduling conditions for the RM and EDF schemes. Recall that $E_i(t)$ is the actual worst-case cost of interferences in jobs of tasks T_1 through T_i in any interval of length t. We let $E_i'(t)$ denote a bound on $E_i(t)$ that is determined using linear programming as described above. For RM scheduling, we have the following.

Theorem 3: *In an RM-scheduled quantum-based uniprocessor system, a set of tasks with objects implemented using the proposed retry algorithms is schedulable if the following holds for every task T_i, where $B_i = \min(Q, \max_{j>i}(c_j'))$.*

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

[5] It is actually possible to eliminate Constraint Set 4, because the linear programming solver will always maximize each f_i^v term to be 1. Furthermore, when substituting 1 for f_i^v in Constraint Set 3, the resulting set of constraints implies those given in Constraint Set 6, so these constraints can be removed as well. We did not minimize the constraint sets in this way because we felt that this would make them more difficult to understand, especially when comparing them against those in [4].

This condition is obtained by modifying one proved in [4] by including a blocking factor for the scheduling quantum. For EDF scheduling, we have the following.

Theorem 4: *In an EDF-scheduled quantum-based uniprocessor system, a set of tasks with objects implemented using the proposed retry algorithms is schedulable if the following holds.*

$$(\forall t :: \sum_{j=1}^{N} \left\lfloor \frac{t}{p_j} \right\rfloor c_j + E_N'(t-1) \leq t) \quad \square$$

This condition was also proved in [4]. Since t is checked beginning at time 0, a blocking factor is not required. As stated, the expression in Theorem 4 cannot be verified because the value of t is unbounded. However, there is an implicit bound on t. In particular, we only need to consider values less than or equal to the least common multiple of the task periods. (If an upper bound on the utilization available for the tasks is known, then we can restrict t to a much smaller range [9].)

Note that, in a quantum-based system, no object access by a task that is guaranteed to complete within the first quantum allocated to a job of that task can be interfered with. Thus, such an access can be performed using a less-costly code fragment that is purely sequential. All of the scheduling conditions presented in this subsection can be improved by accounting for this fact.

In the full paper [1], we show how object-sharing overheads arising from algorithms as proposed here affect lag-bound calculations in proportional-share (PS) scheduled systems. In the PS scheduling literature, the term "client" is used to refer to a schedulable entity. Each client is assigned a *share* of the processor, which represents the fraction of processing time that that client should receive. Quanta are allocated in a manner that ensures that the amount of processor time each client receives is commensurate with its share. The *lag* of a client is the difference between the time a client should have received in an ideal system with a quantum approaching zero, and the time it actually receives in a real system. Stoica et al. showed that optimal lag bounds can be achieved by using *earliest-eligible-virtual-deadline-first* (EEVDF) scheduling [20]. As we show in the full paper, the lag bounds of Stoica et al. can be applied in a system in which our shared-object algorithms are used by simply inflating the cost of a client's request by the cost of one retry loop for every quantum boundary it crosses.

Experimental Comparison. In order to compare the retry-cost estimates produced by the linear programming methods proposed in this paper and in [4], we conducted a series of simulation experiments involving randomly-generated task sets scheduled under the RM scheme. Each task set in these experiments was defined to consist of ten tasks that access up to ten shared objects. 120 task sets

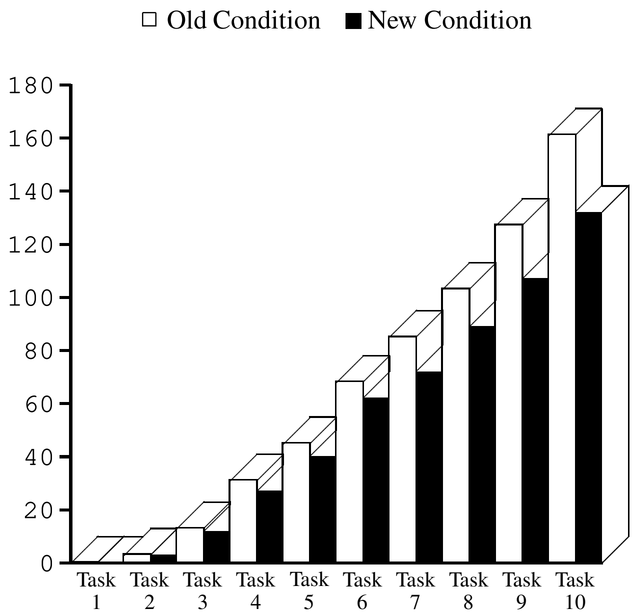

Figure 3. Comparison of linear programming scheduling conditions. Each task's average estimated retry cost is shown.

were generated in total, and for each task set, a retry cost was computed for each task using the two methods being compared. Due to space limitations, the exact methodology we used in generating task sets is not described here; see [1] for details. The results of these experiments are depicted in Figure 3. This figure shows the average retry cost of each task over all generated task sets as computed by each method. As before, tasks are indexed in order of increasing periods. Thus, T_1 has highest priority in all experiments, and as a result, its retry cost is estimated to be zero under both methods. It can be seen in Figure 3 that the method of this paper yields retry-cost estimates for higher-priority tasks that are about 10% to 20% lower than those produced by the method of [4]. In addition to determining retry-cost estimates, we also kept track of how long each schedulability check took to complete. On average, the schedulability check proposed in this paper took 11.7 seconds per task set, while the one proposed in [4] took 235 seconds. This is because of the complicated procedure invoked to compute f_i^v values in the method of [4].

3. Multiprocessor Systems

In this section, we describe a new approach to implementing shared objects in quantum-based multiprocessor systems. Using this retry mechanism, scheduling analysis can be performed on each processor using the uniprocessor scheduling conditions considered in the previous section. In Section 3.1, we describe this retry mechanism in detail. In Section 3.2, we present results from experiments conducted to evaluate our approach.

3.1. Implementing Objects

In a multiprocessor system, a retry mechanism by itself is not sufficient, because a task on one processor may be repeatedly interfered with due to object invocations performed by tasks on other processors. Our approach is to use a retry mechanism in conjunction with a preemptable queue lock. A *queue lock* is a spin lock in which waiting tasks form a queue [17]. Queue locks are useful in real-time systems because waiting times can be bounded. With a preemptable queue lock, a task waiting for or holding a lock can be preempted without impeding the progress of other tasks waiting for the lock. Given such a locking mechanism, any preempted operation can be safely retried. As before, we can appeal to the Preemption Axiom to bound retries, because retries are caused only by preemptions, not by interferences across processors. The Preemption Axiom is still reasonable to assume if we focus on systems with a small to moderate number of processors (the cost an operation depends on the spin queue length, which in turn depends on the number of processors in the system). We believe that it is unlikely that a real-time application would be implemented on a large multiprocessor, and even if it were, it is unlikely that one object would be shared across a large number of processors.

Queue locks come in two flavors: array-based locks, which use an array of spin locations [7, 11], and list-based locks, in which spinning tasks form a linked list [17]. List-based queue locks have the advantage of requiring only constant space overhead per task per lock. In addition, list-based queue locks exist in which all spins are local if applied on multiprocessors either with coherent caches or distributed shared memory [17]. In contrast, with existing array-based locks, spins are local only if applied in a system with coherent caches.

All work known to us on preemptable queue locks involves list-based locks [21, 22]. This is probably due to the advantages listed in the previous paragraph that (non-preemptable) list-based locks have over array-based ones. However, correctly maintaining a linked list of spinning tasks in the face of preemptions is very trick. Wisniewski et al. handle such problems by exploiting a rather non-standard kernel interface that has the ability to "warn" tasks before they are preempted so that they can take appropriate action in time [22]. In the absence of such a kernel interface, list maintenance becomes quite hard, leading to complicated algorithms. For example, a list-based preemptable queue lock proposed recently by Takada and Sakamura requires a total of 63 executable statements [21]. Our preemptable queue lock is an array-based lock and is quite simple, consisting of only 17 lines of code. In addition, all that we require the kernel to do is to set a shared variable whenever a task is preempted indicating that that task is no longer running. As with other array-based locks, our algorithm has linear space overhead per lock and requires coherent caches in order for spins to be local. However, most modern workstation-class multiprocessors have coherent caches. Also, in many applications, most objects are shared only by a relatively small number tasks, so having linear space per lock shouldn't be a severe problem. In any event, these disadvantages seem to be far outweighed by the fact that our algorithm is so simple.

Our algorithm is shown in Figure 4. For clarity, the lock being implemented has been left implicit. In an actual implementation, the shared variables *Tail*, *State*, and *Pred* would be associated with a particular lock and a pointer to that lock would be passed to *acquire_lock* and *release_lock*.

The *State* array consists of $2N$ "slots", which are used as spin locations. A task T_i alternates between using slots i and $i + N$. T_i appends itself onto the end of the spin queue by performing a *fetch_and_store* operation on the *Tail* variable (line 5). It then spins until either it is preempted, its predecessor in the spin queue is preempted, or its predecessor releases the lock (line 9). In a system with coherent caches, this spin is local. If T_i stops spinning because its predecessor is preempted, then T_i takes its predecessor's predecessor as its new predecessor (lines 12-13). If T_i is preempted before acquiring the lock, then (when it resumes execution) it stops spinning and re-executes the algorithm using its other spin location (line 2). Note that the Preemption Axiom ensures that T_i will not be preempted when it re-executes the algorithm. In addition, by the time T_i acquires the lock and then releases it to another task, no task is waiting on either of its two spin locations, i.e., they can be safely reused when T_i performs future lock accesses. Without the Preemption Axiom, correctly "pruning" a preempted task from the spin queue would be much more complicated. (For multiprocessors, the Preemption Axiom can be relaxed to state that a task can be preempted at most once across two consecutive attempts to complete the *same* object call. If our lock algorithm is used by tasks on P processors, then a task that is preempted may have to wait for $P - 1$ tasks on other processors to complete their object calls when it resumes execution. Thus, the Preemption Axiom is tantamount to requiring that the quantum is long enough to contain $P + 1$ consecutive object calls in total on the P processors across which the object is shared.)

We have depicted the algorithm assuming that each task performs its object access as a critical section with interrupts turned off (see lines 14 and 17). Instead, object accesses could be performed using lock-free code, in which case the entire implementation would be preemptable. It can be seen in Figure 4 that the code fragment at lines 5-6 is required to be executed without preemption. This ensures that the predecessor of a preempted task can always be determined. As an alternative to disabling interrupts, if a preemption occurs between lines 5 and 6, then the kernel could roll the preempted task forward one statement when saving its state. This alternative would be necessary in systems in which tasks do not have the ability to disable interrupts.

When a task T_i is preempted while waiting for the lock, the kernel must establish *State*$[slot]$ = PREEMPTED. It is not necessary for the kernel to scan state information per lock to do this. The appropriate variable to update can be determined by having a single shared pointer *Stateptr*$[i]$ for

```
shared variable
    Tail: 0..2N − 1 initially 0;
    State: array[0..2N − 1] of {WAITING, DONE, PREEMPTED}
                                                initially DONE;
    Pred: array[0..N − 1] of 0..2N − 1

private variable                               /* local to task T_p */
    pred: 0..2N − 1;
    slot: {p, p + N} initially p
    /* slot is assumed to retain its value across procedure invocations */

procedure acquire_lock()
1:  while true do                         /* can only loop at most twice */
2:     slot := (slot + N) mod 2N;
3:     State[slot] := WAITING;
4:     disable interrupts;
5:     pred := fetch_and_store(&Tail, slot);  /* join end of spin queue */
6:     Pred[slot mod N] := pred;
7:     enable interrupts;
8:     while State[slot] ≠ PREEMPTED do
9:        while State[slot] = WAITING ∧
                     State[pred] = WAITING do /* spin */ od;
          /*
           *  after the spin, State[slot] = PREEMPTED or
           *  State[pred] = PREEMPTED or State[pred] = DONE
           */
10:       if State[slot] ≠ PREEMPTED then
11:          if State[pred] = PREEMPTED then
12:             pred := Pred[pred mod N]; /* predecessor is preempted */
13:             Pred[slot mod N] := pred;      /* get new predecessor */
             else /* State[pred] = DONE */
14:             disable interrupts;
15:             if State[slot] = WAITING then return /* lock acquired */
                                     else enable interrupts
                fi
             fi
          fi
       od
    od

procedure release_lock()
16: State[slot] := DONE;
17: enable interrupts
```

Figure 4. Preemptable spin-lock algorithm for quantum-based multiprocessors. In this figure, task indices are assumed to range over $\{0, \ldots, N-1\}$.

each task T_i that is used across all locks. Prior to assigning "*State*[*slot*] := WAITING" in line 3, T_i would first update *Stateptr*[i] to point to *State*[*slot*]. By reading *Stateptr*[i], the kernel would know which state variable to update upon a preemption. (If locks can be nested, then multiple *Stateptr* variables would be required per task.)

3.2. Experimental Comparison

We have conducted performance experiments to compare our preemptable queue lock algorithm to a preemptable queue lock presented last year by Takada and Sakamura [21]. Their lock is designated as the "SPEPP/MCS algorithm" in their paper, so we will use that term here (SPEPP stands for "spinning processor executes for preempted processors"; MCS denotes that this lock is derived from one published previously by Mellor-Crummey and Scott [17]). The SPEPP/MCS algorithm was the fastest in the face of preemptions of several lock algorithms tested by Takada and Sakamura. Our experiments were conducted using the Proteus parallel architecture simulator [10]. Using a simulator made it easy to provide the kernel interface needed by each algorithm. The simulator was configured to simulate a bus-based shared-memory multiprocessor, with an equal number of processors and memory modules. The simulated system follows a bus-based snoopy protocol with write-invalidation for cache coherence. Tasks are assigned to processors and are not allowed to migrate. On each processor, tasks are scheduled for execution using a quantum-based round-robin scheduling policy. The scheduling quantum in our simulation was taken to be 10 milliseconds.

Figure 5 presents the results of our experiments. In this figure, the average time is shown for a task to acquire the lock, execute its critical section, and release the lock. These curves were obtained with a multiprogramming level of five tasks per processor, with each task performing 50 lock accesses. The execution cost of the critical section was fixed at 600 microseconds. Each task was configured to perform a noncritical section between lock accesses, the cost of which was randomly chosen between 0 and 600 microseconds. The simulations we conducted indicate that only the number of processors in the system affects relative performance; simulations for different numbers of lock accesses and multiprogramming levels resulted in similar graphs. The curves in Figure 5 indicate that the time taken to acquire the lock in our algorithm is up to 25% less than that for the SPEPP/MCS algorithm (the time taken to acquire the lock is obtained by subtracting the critical section execution time from the values in Figure 5). We also instrumented the code to measure the time taken to acquire the lock in the best case. For each algorithm, the time taken by a task to acquire the lock is minimized when that task is at the head of the spin queue. The best-case time for acquiring the lock was 100 microseconds for our algorithm, and 200 microseconds for the SPEPP/MCS algorithm.

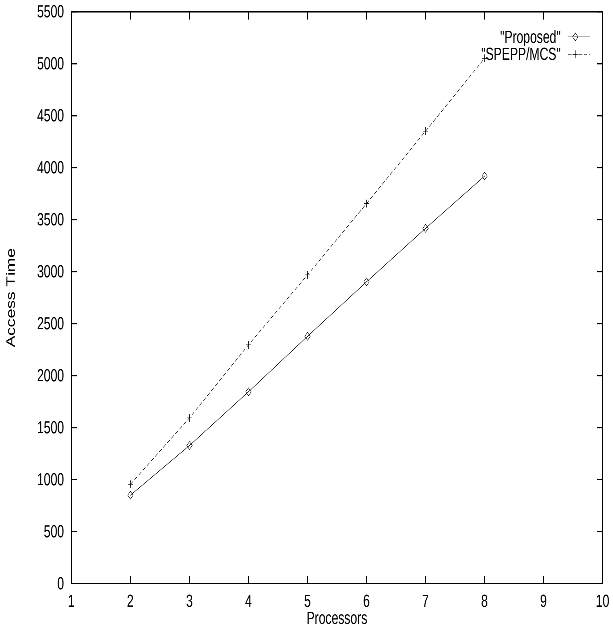

Figure 5. Experimental comparison of preemptable spin-lock algorithms. Curves show average access times (in microseconds).

4. Concluding Remarks

We have presented a new approach to implementing shared objects in quantum-based real-time uniprocessor and multiprocessor systems. In the proposed object implementations, object calls are performed using an optimistic retry mechanism coupled with the assumption that each task can be preempted at most once across two consecutive object calls. We have presented experimental evidence that such implementations should entail low overhead in practice.

In a recent related paper, Anderson, Jain, and Ott presented a number of new results on the theoretical foundations of wait-free synchronization in quantum-based systems [2]. It was shown in that paper that the ability to achieve wait-free synchronization in quantum-based systems is a function of both the "power" of available synchronization primitives and the size of the scheduling quantum. We hope the results of [2] and this paper will spark further research on synchronization problems arising in quantum-based systems.

Acknowledgement: We are grateful to Alex Blate for his help in running simulation experiments. We also acknowledge David Koppelman for his help with the Proteus simulator.

References

[1] J. Anderson, R. Jain, and K. Jeffay Efficient Object Sharing in Quantum-Based Real-Time Systems (expanded version of this paper). Available at http://www.cs.unc.edu/∼anderson/papers.html.

[2] J. Anderson, R. Jain, and D. Ott. Wait-free synchronization in quantum-based multiprogrammed systems. In *Proceedings of the 12th International Symp. on Distributed Computing (to appear)*. Springer Verlag, 1998.

[3] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symp.*, pp. 111–122. 1997.

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

[5] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects in priority-based systems. In *Proceedings of the 16th ACM Symp. on Principles of Distributed Computing*, pp. 229–238. 1997.

[6] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. *ACM Trans. on Computer Systems*, 15(6):388–395, 1997.

[7] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[8] N. C. Audsley, I. J. Bate, and A. Burns. Putting fixed priority scheduling into engineering practice for safety critical applications. In *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symp.*, pp. 2–10, 1996.

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

[10] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, Cambridge, Massachusetts, 1992.

[11] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.

[12] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the USENIX Association Second Symp. on Operating Systems Design and Implementation*, pp. 123–136, 1996.

[13] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th IEEE Symp. on Real-Time Systems*, pp. 129–139. 1991.

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

[15] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw: A solution to a real-time synchronization problem. In *Proceedings of the 14th IEEE Symp. on Real-Time Systems*, pp. 131–137. 1993.

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

[17] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.

[18] S. Ramamurthy. *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 1997.

[19] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal support. In *Proceedings of the 15th ACM Symp. on Principles of Distributed Computing*, pp. 233–242. 1996.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symp.*, pp. 288–299. 1996.

[21] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Proceedings of the 18th IEEE Real-Time Systems Symp.*, pp. 134–143. 1997.

[22] R. Wisniewski, L. Kontothanassis, and M. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proceedings of the Fifth ACM Symp. on Principles and Practices of Parallel Programming*, pp. 199–206. 1995.