Supporting Lock-free Synchronization in Pfair-scheduled Systems ⋆

Philip Holman

Department of Computer Science, University of North Carolina
Chapel Hill, North Carolina 27599-3175 USA

James H. Anderson

Department of Computer Science, University of North Carolina
Chapel Hill, North Carolina 27599-3175 USA

Abstract

We consider various techniques for implementing shared objects and for accounting for object-sharing overheads in Pfair-scheduled multiprocessor real-time systems. Lock-free objects are more economical than locking techniques when implementing relatively simple objects such as buffers, stacks, queues, and lists. In this paper, we explain how to bound the duration of lock-free object accesses under Pfair multiprocessor scheduling. We also show that these durations can be reduced by combining tasks into supertasks, *i.e.*, a group of tasks that are scheduled as a single entity; this is because the use of supertasks can prevent interfering tasks from executing in parallel, thereby reducing the worst-case durations of object accesses. Indeed, we show that supertasking can even enable the use of less costly *uniprocessor* synchronization techniques when all tasks sharing an object reside in the same supertask. We illustrate these optimizations with a case study that focuses on shared queues. Finally, we present and experimentally evaluate a simple heuristic for assigning tasks to supertasks.

Keywords: lock-free, synchronization, Pfairness, real-time, scheduling, multiprocessor, supertasking

⋆ Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, CCR 0204312, and CCR 0309825. Some results in this paper were presented in preliminary form at the 14th Euromicro Conference on Real-time Systems [18].

1 Introduction

There has been much recent work on scheduling techniques that ensure fairness, temporal isolation, and timeliness among tasks multiplexed on a set of processors. Under fair scheduling disciplines, tasks are assigned weights, and each task is scheduled at a rate that is in proportion to its weight. Executing tasks at predictable rates prevents "misbehaving" tasks from exceeding their weight-defined rates (temporal isolation), and allows real-time deadlines to be guaranteed when feasible (timeliness).

In recent years, there has been considerable interest in fair scheduling algorithms for multiprocessor systems [3–5,7–10]. One reason for this interest is the fact that fair scheduling algorithms, at present, are the only known means for optimally scheduling recurrent real-time tasks on multiprocessors. In addition, there has been growing practical interest in such algorithms. Ensim Corp., for example, an Internet service provider, has deployed multiprocessor fair scheduling algorithms in its product line [10].

One limitation of prior work on multiprocessor fair scheduling algorithms is that only *independent* tasks that do not synchronize or share resources have been considered. In contrast, tasks in real systems usually are not independent. Synchronization entails additional overhead, which must be taken into account when determining system feasibility [2,6,26–29]. Unfortunately, prior work on real-time synchronization has been directed at uniprocessor systems, or systems implemented using non-fair scheduling algorithms (or both), and thus cannot be directly applied in fair-scheduled multiprocessor systems. (Indeed, fair-scheduled *uniprocessor* systems were first considered only very recently [9,12,14,22].)

In this paper, we consider the problem of object sharing in fair-scheduled multiprocessor systems. Unfortunately, the use of locking synchronization is problematic in such systems because blocking prevents tasks from executing at a consistent rate, thereby making rate-based scheduling more difficult. For this reason, we consider the use of lock-free techniques, which appear to be better-suited to rate-based scheduling.

Under lock-free synchronization, shared objects are implemented without critical sections or related mechanisms. Instead, operations are optimistically attempted: if an operation fails, then it is simply retried until successful. As a result, tasks never suspend (due to synchronization). Unfortunately, lock-free techniques can only be effectively applied to implement simple shared objects. To support complex objects and device synchronization, lock-based techniques are needed. Support for lock-based synchronization is discussed in [16,19].

Contributions of this paper. We consider the use of lock-free synchronization under multiprocessor fair scheduling. We take as our notion of fairness the *Pfairness* constraint proposed by Baruah *et al.* [7], which provides fairness guarantees while allocating processor time in fixed-size quanta. Although we consider only Pfair scheduling here, the results presented herein exploit the use of a scheduling quantum and hence can be applied to other quantum-based multiprocessor systems as well.

This paper includes three primary contributions. First, we explain how to account for retry overhead under Pfair scheduling, *i.e.*, we derive a bound on the worst-case duration of an object access. Second, we consider the use of supertasking [17,20,24] to reduce the worst-case number of retries experienced by accesses. A *supertask* is merely a collection of tasks, called *component* tasks, that is scheduled as a single entity; when a supertask is scheduled, it selects one of its component tasks for execution. In addition to reducing contention for shared objects, supertasking can also enable the use of less costly uniprocessor lock-free algorithms (as explained later). We use a case study to illustrate this last benefit. Finally, we present and experimentally evaluate a simple heuristic for assigning tasks to supertasks in order to reduce retry overhead.

The rest of the paper is organized as follows. We summarize the basics of Pfair scheduling and supertasking in Section 2. We then introduce lock-free synchronization by illustrating the concept in Section 3. In Section 4, we explain how to bound the duration of a lock-free access, both with and without supertasking. Two implementations of a shared queue are then presented in Section 5 to demonstrate the algorithmic benefits that can be obtained by using supertasks. Our assignment heuristic is then presented in Section 6, followed by an experimental evaluation in Section 7. We conclude in Section 8.

2 Background

In this section, we summarize background information that is related to the results presented herein.

The problem. We consider the scheduling of a task set τ that consists of tasks that access one or more lock-free shared objects on M processors using a Pfair scheduler. We let Γ denote the set of lock-free objects shared by τ. The goal of this work is to determine the maximum duration of each access to each object ℓ ($\in \Gamma$). These duration bounds can then be used when determining whether all timing constraints can be met. We do not assume that tasks access only lock-free objects. Support for other common complexities, including task suspensions and locking synchronization, are discussed in [16].

Pfair scheduling. Under Pfair scheduling, each task T is characterized by a *weight* $T.w$ in the range $(0, 1]$. Conceptually, $T.w$ is the fraction of a single processor to which T is entitled. We let $T = \mathbf{PF}(w)$ to denote a Pfair task with $T.w = w$.

Time is subdivided into a sequence of fixed-length *slots*. To simplify the presentation, we use the slot length as the basic time unit, *i.e.*, slot i corresponds to the time interval $[i, i + 1)$. Within each slot, each processor may be allocated to at most one task. For instance, in Figure 1(b), task B is scheduled in slot 3, which corresponds to the time interval $[3, 4)$. (The rest of this figure is considered in detail below.) Task migration is allowed. We let Q denote the *quantum* size, *i.e.*, the amount of processor time actually provided by each processor within each slot. In a real system, some processor time is unavoidably consumed in each slot by system activities, such as scheduling. We refer to such overhead as *per-slot* overhead. When practical overheads are ignored, as is commonly done in the literature, $Q = 1$.

Pfair scheduling tracks the allocation of processor time in a fluid schedule; deviation is formally expressed as $lag(T, t)$, which is defined below.

$$lag(T, t) = fluid(T, 0, t) - received(T, 0, t) \tag{1}$$

In the above equation, $received(T, t_1, t_2)$ denotes the amount of processor time received by T over $[t_1, t_2)$, while $fluid(T, t_1, t_2)$ denotes the amount of processor time guaranteed by fluid scheduling over this interval. As explained in [16], $fluid(T, t_1, t_2)$ is defined as shown below. [1]

$$fluid(T, t_1, t_2) = T.w \cdot (t_2 - t_1) \cdot Q \tag{2}$$

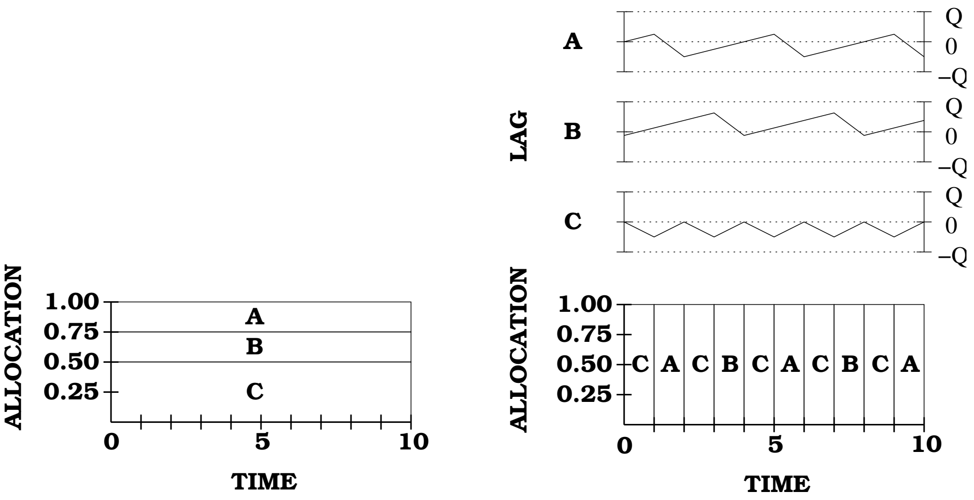
The above formula follows from the fact that each processor provides $(t_2 - t_1) \cdot Q$ units of processor time to tasks over $[t_1, t_2)$. Each task T is then entitled to a fraction $T.w$ of this quantity. (See [16] for a more detailed explanation of fluid scheduling.) Using this notion of lag, the *Pfairness* timing constraint for a task T can be formally defined as shown below.

$$\text{for all } t,\ |lag(T, t)| < Q \tag{3}$$

Informally, T's allocation must always be within one quantum of its fluid allocation.

Figure 1(a) shows ideal (*i.e.*, $Q = 1$) fluid and Pfair uniprocessor schedules for a task set containing three Pfair tasks: $A = \mathbf{PF}(1/4)$, $B = \mathbf{PF}(1/4)$, and $C = \mathbf{PF}(1/2)$. In Figure 1(b), changes in each task's lag are shown across the top of the schedule.

[1] Because $Q = 1$ is commonly assumed, Q typically does not appear in similar formulas in the literature.



(a) Fluid scheduler (b) Lag-based scheduler

Fig. 1. Sample schedules for $\tau = \{A, B, C\}$ where $A = \mathbf{PF}(1/4)$, $B = \mathbf{PF}(1/4)$, and $C = \mathbf{PF}(1/2)$. **(a)** Schedule produced by a fluid scheduler. **(b)** Schedule produced by a Pfair lag-based scheduler.

Baruah *et al.* [7] showed that a schedule satisfying (3) exists on M processors for a set τ of Pfair tasks if and only if the following condition holds.

$$\sum_{T \in \tau} T.w \leq M \tag{4}$$

Subtasks and windows. The use of quantum-based scheduling effectively subdivides each task into a sequence of quantum-length *subtasks*. Scheduling constraints, *e.g.*, (3), have the effect of specifying a *window* of slots in which each subtask must be scheduled. We let T_i denote the i^{th} subtask of task T, and let $\omega(T_i)$ denote the window of that subtask. Figure 2 shows the window within which each subtask of the task $\mathbf{PF}(3/10)$ must execute based on (3). For example, $\omega(T_2) = [3, 7)$. $\omega(T_i)$ extends from T_i's *pseudo-release*,[2] denoted $r(T_i)$, to its *pseudo-deadline*, denoted $d(T_i)$. In Figure 2, $r(T_2) = 3$ and $d(T_2) = 7$. A schedule satisfies Pfairness if and only if each subtask T_i executes in the interval $[r(T_i), d(T_i))$.

Pfair schedulers. Several Pfair algorithms have been proposed, including PF [7], PD [8], PD^2 [5], and EPDF [4,30]. Each of PF, PD, and PD^2 is optimal, *i.e.*, its use will result in a Pfair schedule whenever (4) is satisfied. EPDF has been shown to be optimal only for systems of at most two processors [4]. Despite this, EPDF offers some practical advantages over the optimal algorithms, such as lower scheduling overhead. For our purposes, the choice of scheduling

[2] The "pseudo" prefix avoids confusion with other uses of the terms "release" and "deadline" in the literature. This prefix will be omitted when the proper interpretation is clearly implied.

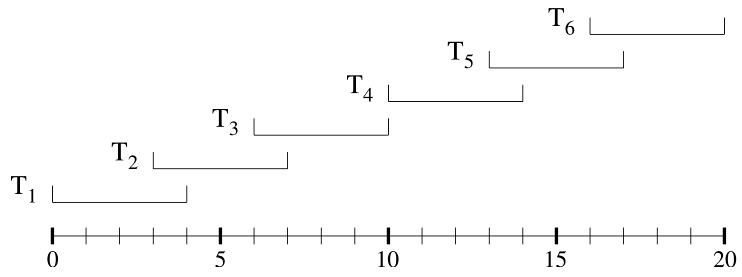


Fig. 2. The first six windows of a task with weight 3/10, as defined by the Pfairness constraint, are shown up to time 20.

algorithm is unimportant, *i.e.*, we focus on producing a scheduler-independent bound on the duration of each lock-free object access. For this reason, we do not provide further detail on the operation of these schedulers. (See the cited papers for additional details.)

Supertasking. In *supertasking* [16,17,20,24], the task set τ is partitioned into a collection of non-empty subsets, π. Each $\mathcal{S} \in \pi$, called a *supertask*, is then assigned a weight $\mathcal{S}.w$ (see below) and competes for the system's processors in place of the tasks in $\mathcal{S}$, called $\mathcal{S}$'s *component tasks*. Whenever $\mathcal{S}$ is scheduled, it selects one of its component tasks for execution. When $|\mathcal{S}| = 1$, the lone component task is assumed to be scheduled directly by the global scheduler. As explained in [24], a supertask would ideally be assigned a weight equal to the cumulative weight of its component tasks. Unfortunately, such a weight is not sufficient in general to ensure that all timing constraints are met.

To see why a supertask with an ideal weight can fail, consider the two-processor Pfair schedule shown in Figure 3. The task set consists of four Pfair tasks ($\mathbf{V} = \mathbf{PF}(1/2)$, $\mathbf{W} = \mathbf{PF}(1/3)$, $\mathbf{X} = \mathbf{PF}(1/3)$, and $\mathbf{Y} = \mathbf{PF}(2/9)$) and one supertask $\mathcal{S}$ that represents the two component tasks $\mathbf{T} = \mathbf{PF}(1/5)$ and $\mathbf{U} = \mathbf{PF}(1/45)$ (shown in the lower region). In Figure 3(a), $\mathcal{S}$ competes with its ideal weight, *i.e.*, $\mathcal{S}.w = 1/5 + 1/45 = 2/9$. All scheduling decisions in the upper region are consistent with the PD^2 policy. In the lower region, allocations within $\mathcal{S}$ are shown, which are made according to the EPDF policy.

As the schedule shows, $\mathbf{T}$ misses a pseudo-deadline at time 10 despite the fact that all pseudo-deadlines are met in the global schedule. This is because no quantum is allocated to $\mathcal{S}$ in the interval $[5, 10)$. Indeed, due to this under-allocation within $[5, 10)$, a deadline miss is unavoidable, regardless of which scheduling policy is used within the supertask. In prior work [16,17,20], we proposed a weight-selection technique for supertasks, called *reweighting*, that ensures the timeliness of component tasks. The cost of such a guarantee is the use of a supertask weight that is higher than the ideal weight, though usually by a reasonably small degree. The difference between the ideal and selected weight is referred to as weight *inflation*.

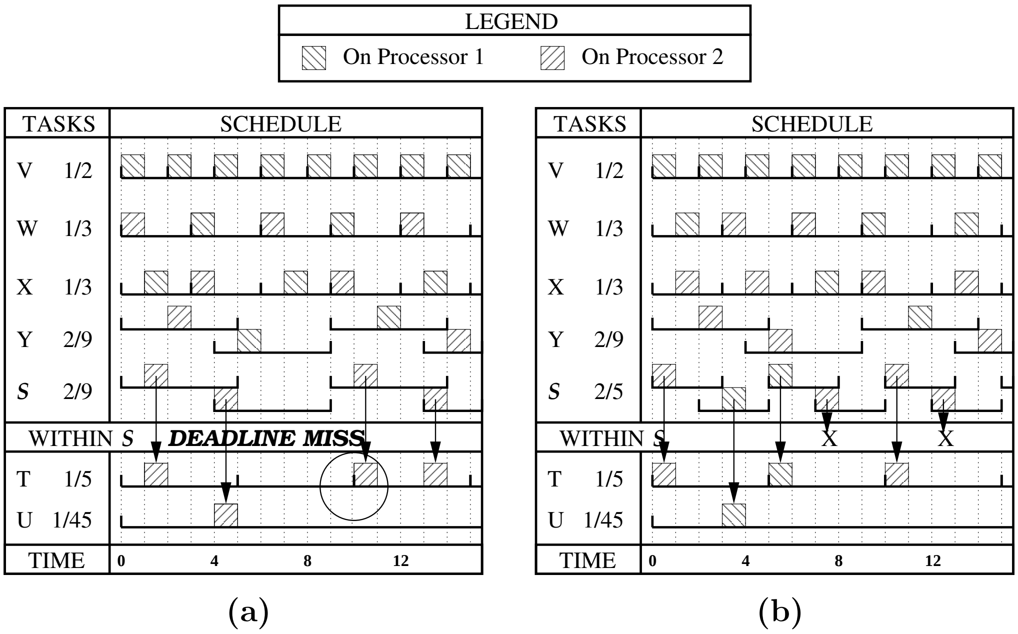


Fig. 3. Two-processor schedule with a **(a)** normal and **(b)** reweighted supertask $\mathcal{S}$.

Figure 3(b) illustrates how reweighting can ensure timeliness. In this schedule, $\mathcal{S}.w$ has been increased to 2/5, resulting in an inflation of $2/5 - 2/9 = 8/45$. (We use an unusually high degree of inflation here to simplify the example.) As shown, no component task violates its timing constraints. However, an unfortunate side effect of reweighting is that a supertask will inevitably be allocated more processor time than its component tasks can utilize; quanta marked with an "X" are allocated to the supertask but cannot be used.

Periodic and sporadic tasks. In the examples and experiments presented later, we consider a task set τ consisting only of periodic [23] and sporadic [25] tasks. (The analysis presented later in the paper does not require the use of a particular task model.) Each periodic and sporadic task T is characterized by four parameters: an *offset* $T.\phi$, a per-job *execution requirement* $T.e$, a *period* $T.p$, and a *relative deadline* $T.d$. Each time the task is invoked, a *job* is released that must complete within $T.d$ time units. The first invocation occurs at time $T.\phi$. Under the periodic (respectively, sporadic) task model, the next invocation occurs exactly (respectively, at least) $T.p$ time units after the previous invocation. Each job requires $T.e$ units of processor time to complete.

We make the simplifying assumption that $T.p = T.d$ for each task T. Under such an assumption, a task T is often characterized by its *utilization* $T.u$, which is defined by $T.e/T.p$. Informally, a task's utilization is the fraction of a single processor's time that will be consumed by the task in the limit. Such tasks can be scheduled under a Pfair scheduler by applying a *mapping rule*, which assigns a weight to each task based on its requirements. Mapping rules for periodic and sporadic tasks are given in [16].

```
typedef Qtype:
    record data: valtype; next: pointer to Qtype
shared var  Head, Tail: pointer to Qtype
private var old, new: pointer to Qtype; input: valtype;
            addr: pointer to pointer to Qtype
procedure Enqueue(input)
    *new := (input, nil);
    do old := Tail;
       if old ≠ nil then addr := &((*old).next)
       else addr := &Head fi
    while ¬CAS2(&Tail, addr, old, nil, new, new)
```

Fig. 4. Lock-free enqueue

3 Lock-free Object Sharing

Lock-free algorithms are an alternative to semaphore-based techniques when implementing software-based shared objects. These algorithms work particularly well for simple objects like buffers, queues, and lists. Lock-free algorithms avoid locking by using *retry loops*. Figure 4 depicts a lock-free enqueue operation with such a loop. In this example, an $\&x$ operation returns the address at which the variable x is stored in memory, while the $*x$ operation returns the data stored at the memory address given by x. The queue is implemented as a singly linked list, where the `next` field acts as the inter-node link.

An item is enqueued by using a *two-word compare-and-swap* (`CAS2`) instruction[3] to atomically update both the tail pointer and either the `next` pointer of the last node or the head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the `CAS2` call succeeds. An important property of lock-free implementations such as this is that operations may *interfere* with each other. In this example, an interference results when a successful `CAS2` call by one task causes another task's `CAS2` call to fail.

We also consider the use of *wait-free* algorithms, which are stricter forms of lock-free algorithms. Wait-freedom strengthens lock-freedom by requiring that each operation eventually makes progress, *i.e.*, starvation must be avoided. To provide such a guarantee, wait-free algorithms must consist only of code segments with a bounded number of loop iterations.

Requirements of analysis. When lock-free objects are used in real-time sys-

[3] The first two parameters of `CAS2` specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed. Although `CAS2` is uncommon, it makes for a simple example here.

tems, bounds on loop retries must be computed to enable scheduling analysis. On uniprocessors, aspects of priority schedulers can be taken into account to determine such bounds [2]. On real-time multiprocessors, lock-free algorithms have long been considered impractical, because the number of interferences *across* processors [4] is difficult to bound. However, as we later show, the predictability of quantum-based scheduling can be exploited to help bound such interferences more tightly than is possible under fully preemptive scheduling.

Limitations. Although lock-free algorithms are ideal for rate-based scheduling, they are not always appropriate. Specifically, lock-free techniques tend to produce efficient implementations only for simple objects, such as queues and buffers, and often rely on the use of strong synchronization primitives, such as the `CAS2` instruction described above. For complex objects, overhead (both time *and* space) can be prohibitive. In such cases, locking synchronization is more suitable. Furthermore, lock-free synchronization can only be applied to software-based shared objects. Locking *must* be used to synchronize accesses to external devices. Hence, support for locking synchronization is a necessity in many systems. Support for locking synchronization under Pfair scheduling is discussed in detail in [16].

4 Analysis

We now describe how to account for lock-free retries in order to bound the duration of an object access. The efficient use of lock-free algorithms is made possible by the quantum-based model underlying Pfair scheduling, as explained below.

4.1 Definitions and Assumptions

Our analysis consists of showing how an access to a lock-free object ℓ can be treated as an independent operation of duration e. This conversion is accomplished by defining e so that it includes not only the actual execution requirement of the access, but also the worst-case retry overhead.

We define the following additional notation to simplify the statement of results:

– Let $A(T, \ell)$ be the maximum number of accesses to object ℓ in a single

[4] An interference is said to occur across processors if the interfering task resides on a different processor than the task experiencing the failed retry-loop attempt.

quantum.[5] When using supertasks, let $A(\mathcal{S}, \ell) = \max\{\ A(T, \ell) \mid T \in \mathcal{S}\ \}$.

- Let $N(\ell)$ denote $\min(M, |\{\ T \mid A(T, \ell) > 0\ \}|)$ when not using supertasks; when using supertasks, let it denote $\min(M, |\{\ \mathcal{S} \mid A(\mathcal{S}, \ell) > 0\ \}|)$. Informally, $N(\ell)$ is the maximum number of processors that may try to access ℓ in a single slot.
- Let $e_B(\ell)$ and $e_R(\ell)$ denote the base and retry overhead of an operation on ℓ, *i.e.*, an operation that retries k times consumes at most $e_B(\ell) + k \cdot e_R(\ell)$ units of processor time.
- Let $e_B^{[m]}(\ell)$ and $e_R^{[m]}(\ell)$ denote the overheads of an m-processor implementation of ℓ. In the analysis considered below, we assume that $e_B(\ell) = e_B^{[N(\ell)]}(\ell)$ and $e_R(\ell) = e_R^{[N(\ell)]}(\ell)$

To simplify the analysis, we consider only a single bound for all operations on a given object ℓ. The analysis presented here is easily extended to support a separate bound for each operation.

We make the following assumptions regarding retries and interferences.

Interference Assumption (IA): Any pair of concurrent accesses to the same object may potentially interfere with each other.
Retry Assumption (RA): A retry can be caused *only* by the completion of an operation on the same object. (Hence, the number of retries is at most the number of concurrent accesses to the object.)
Preemption Assumption (PA): Each operation spans at most two quanta and hence can be preempted at most once.

(PA) stems from the observation that lock-free loop iterations are typically very short relative to Q [1]. Determining whether this assumption holds is straightforward. When (PA) does not hold for an object, lock-free techniques do not provide an efficient means for implementing that object.

Some additional notation is defined below.

- Let $I(T, \ell) = \mathsf{maxsum}_{M-1}\{\ A(U, \ell) \mid U \in \tau/\{T\}\ \}$. Informally, $I(T, \ell)$ is the maximum number of retries that T can experience in a single quantum without supertasking.
- Let $I(\mathcal{S}, \ell) = \mathsf{maxsum}_{M-1}\{\ A(\mathcal{T}, \ell) \mid \mathcal{T} \in \pi/\{\mathcal{S}\}\ \}$. Informally, $I(\mathcal{S}, \ell)$ is the maximum number of retries that $T \in \mathcal{S}$ can experience in a single quantum.

[5] *Timing analysis* is needed to obtain these bounds. This analysis focuses on deriving bounds on the time required to execute each code block in a source listing.

4.2 Analysis without Supertasks

In this subsection, we consider the conversion of object accesses when supertasks are not used.

Theorem 1 *When supertasking is not used, an access to lock-free object ℓ by task T requires at most $e_B(\ell) + (2 \cdot I(T, \ell) + 1) \cdot e_R(\ell)$ units of processor time.*

Proof. By (PA), a single access to ℓ is preempted at most once before completion. In the worst case, this access experiences the maximum number of retries during the quantum preceding the preemption and during the quantum in which the access completes. By (RA), the worst-case number of retries in a single quantum is bounded by the number of concurrent accesses to ℓ within that quantum. Since there are at most $M - 1$ tasks executing in parallel with T's job and each such task U makes at most $A(U, \ell)$ accesses to ℓ within the quantum, it follows that $I(T, \ell)$ is an upper bound on the number of retries that T will perform in each quantum due to parallel interference. (Note that (PA) implies that $I(T, \ell) \cdot e_R(\ell)$ is much smaller than Q.) Therefore, at most $2 \cdot I(T, \ell) + 1$ retries are performed before the access completes. (At most $I(T, \ell)$ retries are needed for each quantum in which T executes. In addition, if T is preempted during an attempt, then an additional retry may be needed due to accesses performed while T was not executing.) Therefore, $e_B(\ell) + (2 \cdot I(T, \ell) + 1) \cdot e_R(\ell)$ is an upper bound on the total processor time required. □

4.3 Analysis with Supertasks

Supertasking can improve performance in the following two ways.

- A supertask can prevent potentially-interfering tasks from executing in parallel. By doing so, the number of retries needed to complete an operation can be reduced.
- By (PA), if all tasks that share an object are component tasks of the same supertask, then a retry is necessary only if the access is preempted. As a result, a simpler wait-free algorithm can often be used.

When supertasks are used, the worst-case number of interferences experienced in a single quantum changes from $I(T, \ell)$ to $I(\mathcal{S}, \ell)$, where $T \in \mathcal{S}$. In addition, algorithmic gains may be obtained by a reduction in $N(\ell)$, which impacts $e_B(\ell)$ and $e_R(\ell)$. As observed earlier, when $N(\ell) = 1$, ℓ can be implemented using a uniprocessor algorithm. We demonstrate the benefit of such an implementation later with a case study.

Theorem 2 *When supertasking is used, an access to lock-free object ℓ by task $T(\in \mathcal{S})$ requires at most $e_B(\ell) + (2 \cdot I(\mathcal{S}, \ell) + 1) \cdot e_R(\ell)$ units of processor time.*

Proof. The proof is virtually identical to that of Theorem 1. However, since T can be scheduled in parallel with at most one component task from each of the other supertasks, the worst-case number of retries that can occur during each quantum is $I(\mathcal{S}, \ell)$ instead of $I(T, \ell)$. □

4.4 Examples

We now use the above theorems to derive weights for a sample task set in which all shared objects use lock-free implementations. To simplify the example, we assume that each task T is a synchronous (*i.e.*, $T.\phi = 0$) periodic task with an integer period that equals the task's relative deadline. We let $\mathcal{J}(T, \ell)$ denote the number of accesses made to the lock-free object ℓ by each job of T. Also, we let $Q = 1$ for this example. Under these assumptions, the weight assignment $T.w = \lceil T.e \rceil / T.p$ is sufficient to guarantee timeliness [16].

Consider the example four-processor task set shown in Figure 5(a)–(b). The value given in the e column is the total execution requirement of each job *without* considering lock-free object accesses. This example considers only two implementations for each object ℓ: a uniprocessor implementation (used when $N(\ell) = 1$) and a multiprocessor implementation (used when $N(\ell) > 1$). We use the above theorems to select task weights for this set below.

Without supertasks. Figure 5(c) summarizes the values that are computed when applying Theorem 1. We explain each column in turn by stepping through the computation of T_{10}'s weight. First, we must bound the number of retries for each of T_{10}'s object accesses. Consider ℓ_1. Applying the definition of $I(T_{10}, \ell_1)$ yields $I(T_{10}, \ell_1) = \mathsf{maxsum}_{M-1}\{\ A(U, \ell_1) \mid U \in \tau / \{T_{10}\}\ \} = \mathsf{maxsum}_3\{2, 1, 1, 1, 1, 0, 0, 0, 0\} = 4$. This is shown in the column labeled $I(\ell_1)$ in Figure 5(c). Next, we compute the worst-case execution cost of a single access to object ℓ_1 by T_{10}, labeled $\lambda(\ell_1)$ in the table. Since $N(\ell_1) = \min(4, 6) = 4$, $e_B(\ell_1) = e_B^{[M]}(\ell_1) = 0.05$ and $e_R(\ell_1) = e_R^{[M]}(\ell_1) = 0.16$. From these values and the expression $\lambda_{T_{10}}(\ell_1) = e_B(\ell_1) + (2 \cdot I(T_{10}, \ell_1) + 1) \cdot e_R(\ell_1)$, we get $\lambda_{T_{10}}(\ell_1) = 0.05 + (2 \cdot 4 + 1) \cdot 0.16 = 1.49$. We can now determine the total execution overhead of T_{10}'s accesses to ℓ_1 in a single job, labeled $\Lambda(\ell_1)$ in the table, by simply multiplying the value in the $\lambda(\ell_1)$ column by that in the $\mathcal{A}(\ell_1)$ column. Doing so yields $\Lambda_{T_{10}}(\ell_1) = 5 \cdot 1.49 = 7.45$. In a similar manner, it can be shown that $\Lambda_{T_{10}}(\ell_2) = 12.084$. Hence, the execution requirement of an entire job of T_{10} is upper-bounded by $50 + 7.45 + 12.084 = 69.534$. Applying the mapping rule given earlier to T_{10} then yields the weight $\lceil 69.534 \rceil / 700 = 70/700$,

T	e	p, d	$\mathcal{A}(\ell_1)$	$A(\ell_1)$	$\mathcal{A}(\ell_2)$	$A(\ell_2)$
T_1	10	100	2	1	0	0
T_2	15	100	1	1	0	0
T_3	15	100	0	0	1	1
T_4	25	100	0	0	2	2
T_5	25	200	2	1	1	1
T_6	30	200	1	1	0	0
T_7	20	200	0	0	1	1
T_8	40	300	3	2	0	0
T_9	65	500	0	0	2	1
T_{10}	50	700	5	2	12	3

(a)

ℓ	$e_B^{[1]}(\ell)$	$e_R^{[1]}(\ell)$	$e_B^{[M]}(\ell)$	$e_R^{[M]}(\ell)$
ℓ_1	0.012	0.08	0.05	0.16
ℓ_2	0.005	0.075	0.017	0.11

(b)

T	$I(\ell_1)$	$I(\ell_2)$	$\lambda(\ell_1)$	$\lambda(\ell_2)$	$\Lambda(\ell_1)$	$\Lambda(\ell_2)$	w
T_1	5	6	1.81	1.447	3.62	0.0	14/100
T_2	5	6	1.81	1.447	1.81	0.0	17/100
T_3	5	6	1.81	1.447	0.0	1.447	17/100
T_4	5	5	1.81	1.227	0.0	2.454	28/100
T_5	5	6	1.81	1.447	3.62	1.447	30/200
T_6	5	6	1.81	1.447	1.81	0.0	32/200
T_7	5	6	1.81	1.447	0.0	1.447	22/200
T_8	4	6	1.49	1.447	4.47	0.0	45/300
T_9	5	6	1.81	1.447	0.0	2.894	68/500
T_{10}	4	4	1.49	1.007	7.45	12.084	70/700

(c)

Fig. 5. **(a)** A sample task set τ on $M = 4$ processors with two lock-free objects ℓ_1 and ℓ_2. All parameters are expressed in units of quanta. **(b)** Parameters of ℓ_1 and ℓ_2. **(c)** Summary of computed values when applying Theorem 1. The λ and Λ columns show selected temporary values.

$\mathcal{S}$	**Components**	$A(\ell_1)$	$A(\ell_2)$	$I(\ell_1)$	$I(\ell_2)$
$\mathcal{S}_1$	T_1, T_2,	2	0	2	3
	T_6, T_8				
$\mathcal{S}_2$	T_3, T_4, T_5,	2	3	2	0
	T_7, T_9, T_{10}				

(a)

T	$I(\ell_1)$	$I(\ell_2)$	$\lambda(\ell_1)$	$\lambda(\ell_2)$	$\Lambda(\ell_1)$	$\Lambda(\ell_2)$	w
T_1	2	3	0.85	0.53	1.70	0.0	12/100
T_2	2	3	0.85	0.53	0.85	0.0	16/100
T_3	2	0	0.85	0.08	0.0	0.08	16/100
T_4	2	0	0.85	0.08	0.0	0.16	26/100
T_5	2	0	0.85	0.08	1.70	0.08	27/200
T_6	2	3	0.85	0.53	0.85	0.0	31/200
T_7	2	0	0.85	0.08	0.0	0.08	21/200
T_8	2	3	0.85	0.53	2.55	0.0	43/300
T_9	2	0	0.85	0.08	0.0	0.16	66/500
T_{10}	2	0	0.85	0.08	4.25	0.96	56/700

(b)

Fig. 6. **(a)** Parameters computed for analysis when using the partitioning $\pi = \{\{T_1, T_2, T_6, T_8\}, \{T_3, T_4, T_5, T_7, T_9, T_{10}\}\}$. **(b)** Summary of all values computed when applying Theorem 2.

as shown in the last column of the table. Since the calculated weights of all tasks are upper-bounded by unity and their sum is upper-bounded by M, (4) implies that this task set is schedulable under any optimal Pfair scheduling policy.

With supertasks. Figure 6(a) shows one possible partitioning of the example task set from Figure 5(a). This partitioning assigns all tasks accessing ℓ_2 to $\mathcal{S}_2$ and then assigns all remaining tasks to $\mathcal{S}_1$. By doing so, we permit the use of the more efficient uniprocessor algorithm for ℓ_2 in place of the multiprocessor version. Therefore, $e_B(\ell_2) = e_B^{[1]}(\ell_2) = 0.005$ and $e_R(\ell_2) = e_R^{[1]}(\ell_1) = 0.075$. However, notice that the multiprocessor version of ℓ_1's algorithm must still be used. Hence, $e_B(\ell_1) = e_B^{[M]}(\ell_1) = 0.05$ and $e_R(\ell_1) = e_R^{[M]}(\ell_1) = 0.16$. Since the computations in Theorem 2 are similar to those already demonstrated, we do not explain them here. The values resulting from each step can be seen in Figure 6(b). Notice that many of the task weights are smaller than in the previous example. In addition, the total weight has reduced from 1.57 to 1.45. Unfortunately, as discussed earlier in Section 2, the use of supertasks produces some loss, which somewhat offsets this improvement.

Remarks. In a later section, we present experimental results that suggest that supertasking often improves schedulability, even with reweighting overhead. However, in a system with relatively few processors, the benefit of supertasking is often negated by its cost. This is because of the limited parallelism in such systems. Recall that the most significant component of the retry overhead is caused by the parallel execution of tasks. Hence, this overhead scales with M and may be insignificant for small values of M. For this reason, it is always advisable to calculate the weights of task sets both with and without supertasks in order to determine whether supertasking is beneficial.

5 Case Study: Queues

In this section, we illustrate the algorithmic benefits that can be obtained by enabling the use of a uniprocessor object implementation instead of a multiprocessor implementation. This is accomplished by presenting two lock-free queue implementations and comparing the number of primitive operations performed by each. These queue implementations are presented only to demonstrate that simpler lock-free implementations often suffice on uniprocessors, *i.e.*, we do not claim that these are the best lock-free queue implementations.

5.1 Variable Tagging

Lock-free algorithms often use the *compare-and-swap* (`CAS`) primitive to update shared variables. `CAS` is similar to the `CAS2` primitive (used in Figure 4), but accesses only one memory location. Unlike `CAS2`, `CAS` (and similar primitives) are fairly common. The primary benefit of `CAS` is that it can avoid the *late-write* problem. This problem occurs when a task T's next instruction updates a shared variable, but another update on that variable occurs first. If T's update succeeds, then the result of the earlier update will be overwritten.

The late-write problem. To illustrate why late writes can be undesirable, consider a simple shared counter that supports only an increment operation. To perform the increment, a task must make a local copy of the current value, increment that local value, and then overwrite the shared value with the local value. Now, suppose two tasks simultaneously invoke the increment operation and read the value 2 from the shared variable, resulting in a local value of 3. Once the first operation updates the counter, the second operation becomes invalid due to the fact that its update is based on an out-of-date counter value (*i.e.*, the increment is based on the counter value 2, but the current value is 3). Hence, the second update, which is called an *enabled late write*, must fail to ensure correctness.

Preventing late writes. By using `CAS`, we can ensure that the value being overwritten is equal to the value read by the task. Such a guarantee would be sufficient for the above example because the shared variable's value monotonically increases;[6] hence, each value is uniquely associated with only one successful increment operation. Unfortunately, when multiple operations may successfully write the same value, simply comparing the current value to that read earlier is not sufficient to determine whether other updates occurred after that read.

The `CAS` primitive can be used to detect modification by subdividing a word of memory into two sets of bits. One set stores the current value of the variable, while the second stores a counter like that discussed above. This counter field is called the variable's *tag*. Each time an update is attempted, an increment operation is applied to the tag. By the same reasoning given above, each value of the tag is uniquely associated with a single operation. Hence, updates of the variable can be detected by checking the tag. Since the tag and value comprise a single word of memory, a `CAS` operation can be used to atomically compare and update both values simultaneously. Both implementations presented here use tagged variables.

template *tagged*(T): **record** *tag*: **integer**; *value*: T

Our algorithms use the template definition shown above to define tagged variables. As stated above, each tagged variable is assumed to require only one memory word. Though accounting for the bounded range of the tags is an important issue, we ignore this complication here since such advanced issues are outside the scope of this discussion. Instead, we make the simplifying assumption that tag ranges are unbounded.

5.2 Algorithms

Our multiprocessor and uniprocessor queue implementations are shown in Figures 8 and 9, respectively. Since we are introducing this algorithm only to illustrate a point, we omit a proof of correctness. (The proof can be found

[6] To simplify the discussion, we ignore the possibility of rollover.

in [16].) Both implementations are based on a linked list, in which each list node consists of a value field, *value*, and a tagged pointer to the next list node, *next*. In addition, the tagged pointers *Head* and *Tail* record the start and end, respectively, of the list. To allow safe concurrent enqueue and dequeue operations, the linked list contains a *dummy* head node. The structure of the queue is illustrated in Figure 7.

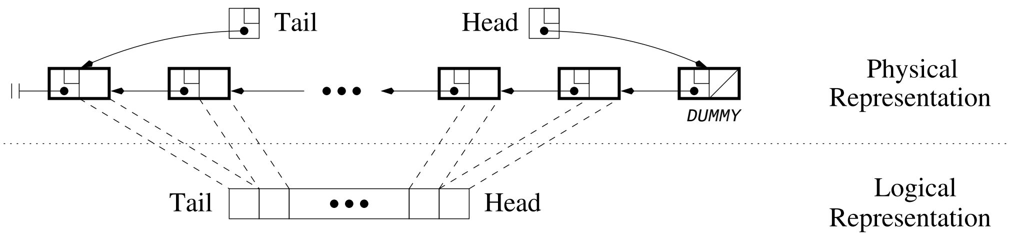


Fig. 7. Physical and logical representation of the shared queue.

The *next* and *Tail* pointers are tagged to support multiple enqueuers (writers). For the single-enqueuer case, the enqueue procedure simplifies to `Enqueue_1W`, shown below `Enqueue_MW` in Figure 8, and these tag fields can be safely removed. Similarly, the tag field in *Head* supports multiple dequeuers (readers) and can be safely removed in the single-dequeuer case. In this case, `Dequeue_MR` simplifies to the procedure `Dequeue_1R` in Figure 8. `Enqueue_1W` and `Dequeue_1R` are common to both implementations, and thus are not repeated in Figure 9.

Multiprocessor algorithm. In the multiprocessor version, an enqueue operation invokes `Enqueue_MW` and passes a node (*in*) containing the value to be enqueued. The *next* field is initialized in lines 1–2 and the *done* flag cleared in line 3. The *next* field of the last node is then read in lines 4–5. If the comparison at line 6 fails, then another enqueue has completed and a retry occurs. Otherwise, an update of the *next* field is attempted (line 7). If successful, then the new node has been chained onto the end of the list and it remains only to update *Tail*. This is done in lines 8–9. If the `CAS` at line 7 is not successful, then another node, call it X, has already been chained onto the end of the list by another task, in which case the operation must be retried. Lines 8–9 ensure that *Tail* is correctly updated before the retry is performed. (The task enqueuing X may not have updated *Tail* yet.)

The `Dequeue_MR` procedure returns either a pointer to a node that contains the dequeued value or *nil* to signify an empty queue. Since *Head* always points to a dummy node, the node *following* the dummy node, if one exists, contains the value at the head of the queue. Thus, `Dequeue_MR` seeks to remove the dummy head node and return the data stored in the node *after* it. This latter node then becomes the new dummy head node. The operation begins by initializing the *done* flag at line 17 and then reading address of the dummy node at line 18. This address is compared to *Tail* at line 19. If equal, then either the list is

```
template tagged(T):
    record
        tag: integer;
        value: T

typedef node:
    record
        value: element;
        next: tagged(pointer to node)

private var
    x, h, t: tagged(pointer to node);
    done: boolean; out: element;
    in: pointer to node

procedure Enqueue_MW(in)
1:  x := *(in).next;
2:  *(in).next := (x.tag+1,nil);
    do
3:      done := false;
4:      t := Tail;
5:      x := *(t.value).next;
6:      if t = Tail then
7:          done := CAS(&(*(t.value).next),
                  (x.tag,nil),(x.tag+1,in));
8:          x := *(t.value).next;
9:          CAS(&Tail, t, (t.tag+1,x.value))
        fi
10: while ¬done

procedure Enqueue_1W(in)
11: x := *(in).next;
12: *(in).next := (x.tag+1,nil);
13: t := Tail;
14: x := *(t.value).next;
15: *(t.value).next := (x.tag+1,in);
16: Tail := (t.tag+1,in)
```

```
shared var
    Head, Tail: tagged(pointer to node)

procedure Dequeue_MR()
    returns pointer to node
    do
17:     done := false;
18:     h := Head;
19:     if h.value = Tail.value then
20:         if h = Head then
21:             return nil
            fi
        else
22:         x := *(h.value).next;
23:         if x.value ≠ nil then
24:             out := *(x.value).value;
25:             done := CAS(&Head, h,
                        (h.tag+1,x.value))
            fi
        fi
26: while ¬done;
27: *(h.value) := (out,(x.tag+1,nil));
28: return h.value

procedure Dequeue_1R()
    returns pointer to node
29: h := Head;
30: if h.value = Tail.value then
31:     return nil
    else
32:     x := *(h.value).next;
33:     out := *(x.value).value;
34:     Head := (h.tag+1,x.value);
35:     *(h.value) := (out,(x.tag+1,nil));
36:     return h.value
    fi
```

Fig. 8. Multiprocessor (lock-free) shared queue.

empty or the node referenced by h has been dequeued and re-enqueued by other concurrent operations. In the latter case, $h \neq Head$ must hold due to tagging. Thus, line 20 correctly distinguishes between these possibilities. If the test at line 19 is not successful, then either an interference has occurred, or the list is non-empty. In either case, an attempt is made to dequeue the head node in lines 22-25. (If an interference has indeed occurred, then this dequeue attempt will fail. Attempting the operation is the simplest way to detect the interference.) If $x.value$ is nil at line 23, then an interference has occurred, and the operation is retried. In line 24, the data value in the node after the dummy node is read, as explained above. In line 25, an attempt is made to advance the *Head* pointer past the (old) dummy node. If this attempt fails, then the operation is retried. (If the `CAS` at line 25 succeeds, then no other enqueue could have completed between lines 18 and 25.) Line 27 simply stores the dequeued value into the removed node so that it can be returned at line 28.

Uniprocessor algorithm. The uniprocessor wait-free queue implementation is obtained by unrolling the loop in the multiprocessor version a constant number of times and then simplifying the code listing. The result is shown in

```
template tagged(T):
    record
        tag: integer;
        value: T

procedure Enqueue_MW(in)
1:  pm := false;
2:  x := *(in).next;
3:  *(in).next := (x.tag+1,nil);
4:  t := Tail;
5:  x := *(t.value).next;
6:  if x.value ≠ nil then
7:      if CAS(&Tail, t,
                (t.tag+1,x.value)) then
8:          t := (t.tag+1,x.value);
9:          x := *(t.value).next;
10:         pm := (x.value ≠ nil)
        else
11:         pm := true
        fi
    fi;
12: if ¬pm ∧ t = Tail then
13:     if CAS(&(*(t.value).next),x,
                (x.tag+1,in)) then
14:         CAS(&Tail, t, (t.tag+1,in))
        else
15:         pm := true
        fi
    fi;
16: if pm then
17:     t := Tail;
18:     x := *(t.value).next;
19:     if x.value ≠ nil then
20:         Tail := (t.tag+1,x.value);
21:         t := (t.tag+1,x.value);
22:         x := *(t.value).next
        fi;
23:     *(t.value).next := (x.tag+1,in);
24:     Tail := (t.tag+1,in)
    fi
```

```
typedef node:
    record
        value: element;
        next: tagged(pointer to node)

shared var
    Head, Tail: tagged(pointer to node)

private var
    x, h, t: tagged(pointer to node);
    in: pointer to node; out: element;
    pm: boolean

procedure Dequeue_MR()
    returns pointer to node
25: h := Head;
26: if h.value = Tail.value then
27:     if h = Head then
28:         return nil
        fi
    else
29:     x := *(h.value).next;
30:     if x.value ≠ nil then
31:         out := *(x.value).value;
32:         if CAS(&Head, h,
                    (h.tag+1,x.value)) then
33:             *(h.value) := (out,(x.tag+1,nil));
34:             return h.value
            fi
        fi
    fi;
35: h := Head;
36: if h.value = Tail.value then
37:     return nil
    else
38:     x := *(h.value).next;
39:     out := *(x.value).value;
40:     Head := (h.tag+1,x.value);
41:     *(h.value) := (out,(x.tag+1,nil));
42:     return h.value
    fi
```

Fig. 9. Uniprocessor (wait-free) shared queue.

Figure 9.

The loop in `Enqueue_MW` (Figure 8) is unrolled three times to produce the procedure shown in Figure 9. The need for three loop iterations is illustrated by task B in Figure 10. In this scenario, task A begins an operation close to the slot boundary and is preempted between lines 8 and 9. As a result, it leaves its node linked to the tail of the queue (see the t_A state shown in the lower portion of Figure 10). Suppose no other operations occur until B initiates its operation. Because the *next* pointer in the node referenced by *Tail* is not *nil* (*i.e.*, it references A's node), the first `CAS` call in line 7 fails and line 9 updates *Tail* to reference A's node. B is then preempted immediately before line 7 during the second iteration, and some other task (C) performs an operation while B is preempted. In this case, B's second execution of line 7 also fails (when B resumes) because the information stored in x and t is out-of-date. Hence, a third iteration is performed. This last iteration is guaranteed

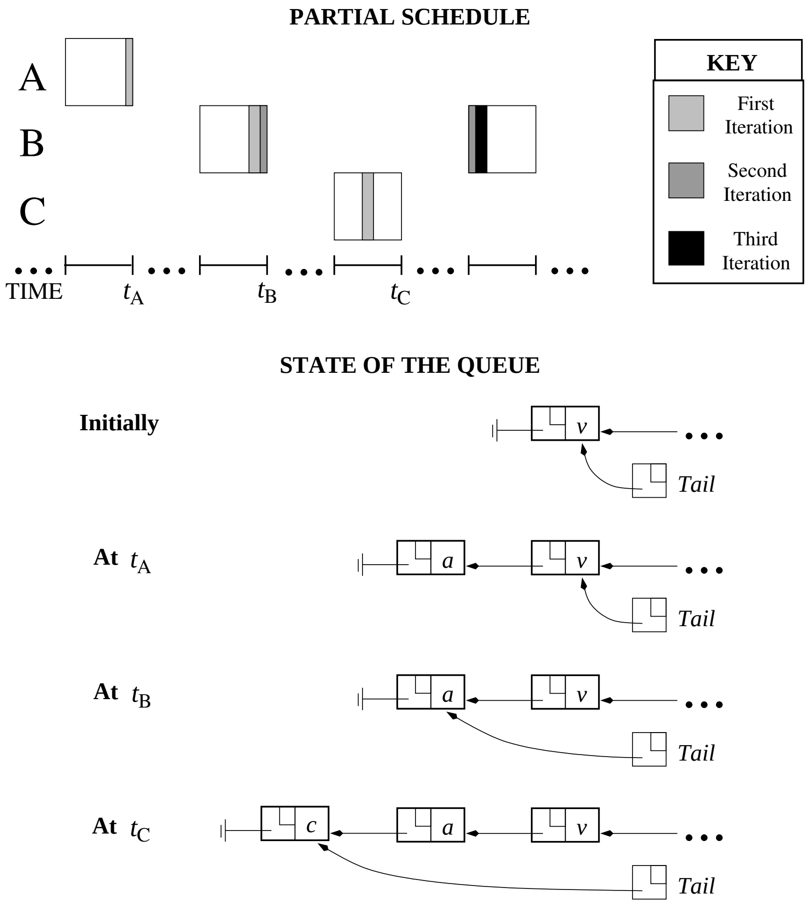


Fig. 10. Illustration of the worst-case scenario when using the multiprocessor lock-free queue implementation on a quantum-based uniprocessor. This same worst case applies to the use of this implementation within a supertask.

to succeed. It follows that three loop iterations occur in the worst case.[7]

In Figure 9, the first loop iteration, which checks for a partially completed operation, produces lines 6–11. The second iteration, which attempts to perform the enqueue operation, produces lines 12–15. If preemption occurs during either of these two "iterations," then (PA) ensures that the next iteration is effective non-preemptable. Therefore, the third iteration can be simplified, as shown in lines 16–24.

By (PA), the multiprocessor version of `Dequeue_MR` iterates at most twice on a uniprocessor. The first loop iteration corresponds to lines 25–34 in Figure 9. If the operation is preempted, then a second iteration is performed, which corresponds to lines 35–42.

[7] Notice that only one of these iterations was caused by a preemption. Thus, although there are three loop iterations, there is only one retry. Hence, this scenario is not a violation of (PA).

M	Procedure	Path	R	W	CAS	$1{\times}R$	$1{\times}W$	$3.5{\times}CAS$	Total
> 1	`Enqueue_MW`		13	1	6	13	1	21	35
> 1	`Dequeue_MR`		10	2	2	10	2	7	19
1	`Enqueue_MW`	pm	8	4	2	8	4	7	19
1	`Enqueue_MW`	$\neg pm$	5	1	3	5	1	10.5	16.5
1	`Dequeue_MR`		8	3	1	8	3	3.5	14.5

Fig. 11. Shared-memory instruction counts along worst-case code paths for both queue algorithms.

5.3 Comparison

One simple method of comparing the relative efficiency of these algorithms is to count the number of shared-memory reads and writes (denoted R and W, respectively) and the number of `CAS` invocations (denoted CAS) that occur in the worst case. This comparison must necessarily be made on a uniprocessor. For the multiprocessor version of `Enqueue_MW`, the worst-case path under uniprocessor execution includes *three* loop iterations, with six `CAS` calls being made in lines 7 and 9. In addition, lines 1–2 contribute one read and write, and each loop iteration contributes four reads. Therefore, in the worst case, the multiprocessor `Enqueue_MW` algorithm has $R = 13$, $W = 1$, and $CAS = 6$.

A similar analysis can be applied the other procedures with the exception of `Enqueue_MW` in Figure 9. It is unclear which code path produces the worst-case behavior in this procedure. For this reason, we consider two paths through the code. Figure 11 summarizes the instruction counts along all considered paths.

The cost of synchronization primitives. Synchronization primitives usually require more cycles than uncached reads and writes. LaMarca [21] noted that the *load-linked/store-conditional* instruction pair (which provides functionality similar to `CAS`) on a DEC 3000-400 with a 130 MHz Alpha 2 21064 CPU requires approximately 3.5 times the number of cycles as an uncached shared-memory read. Though it is not a modern processor, this and other older architectures are still being used today, particularly in embedded systems.

Comparison. A simple method for comparing these algorithms is to calculate the weighted sums of the previous instruction counts based on LaMarca's observations. Though this comparison is far from exact, it does provides some insight into the relationship between these implementations. Figure 11 shows the sums produced by the worst-case code paths. These sums suggest that using the uniprocessor algorithm provides an improvement in `Enqueue_MW` and `Dequeue_MR` of around 84% ($\frac{35}{19} \approx 1.84$) and 31% ($\frac{19}{14.5} \approx 1.31$), respectively.

Improvement trends. For lock-free operations that update only one variable, a uniprocessor implementation may provide only a marginal improvement. However, for more complex operations that update multiple variables, significant improvement is likely. This is demonstrated by `Enqueue_MW`, which shows significant improvement due to the need to update two variables.

Operations that perform multiple updates can be greatly simplified by using a *multi-word compare-and-swap* (`MWCAS`). This primitive generalizes both `CAS` and `CAS2` by making the number of words involved in the operation an argument. Though it is impractical to provide a `MWCAS` primitive in hardware, it *can* be efficiently implemented in software on a uniprocessor [1]. In contrast, no efficient implementation is known for multiprocessors. For this reason, the class of objects that have efficient uniprocessor lock-free implementations is far larger than the class that can be efficiently implemented on multiprocessors.

6 Assigning Tasks to Supertasks

In this section, we present a simple heuristic for assigning tasks to supertasks in order to reduce lock-free overhead. In the next section, we present the results of an experimental study that compares the overhead experienced under several approaches, including the use of the heuristic presented here. The purpose of this section and the experimental study is to demonstrate that supertasking can be an effective means of reducing lock-free overhead, even when reweighting overhead is considered. Since reducing lock-free overhead is but one benefit of supertasking, the heuristic considered here is too simplistic for most systems. Indeed, provisioning supertasks in order to minimize the total overhead remains a prominent unsolved optimization problem.

6.1 The Heuristic

Since supertask assignment is a variation of the partitioning problem, which is known to be NP-hard in the strong sense [15], we consider the use of a heuristic. A pseudo-code version of our heuristic is shown in Figure 12. Tasks are prioritized based upon the number of accesses made to each object and the degree of contention for that object. We define the *weighted contention* for object ℓ by the value $e_R^{[M]}(\ell) \cdot \sum_{T \in \tau} \frac{\mathcal{J}(T,\ell)}{T.p}$. In this expression, $\sum_{T \in \tau} \frac{\mathcal{J}(T,\ell)}{T.p}$ gives the frequency of accesses to ℓ by all tasks in τ. $e_R^{[M]}(\ell)$ then represents the interference penalty. All tasks that access the object ℓ with the highest weighted contention are assigned to supertasks first. Since each task T's interference depends on $A(T,\ell)$, tasks are assigned in non-increasing order by $A(T,\ell)$.

Create $|\tau|$ empty supertasks and add them to π;
$\tau' := \tau$;
$\Gamma' := \Gamma$;
while $|\Gamma'| > 0 \wedge |\tau'| > 0$ **do**
  Select $\ell \in \Gamma'$ with largest $e_R^{[M]}(\ell) \cdot \sum_{T \in \tau} \frac{\mathcal{I}(T,\ell)}{T.p}$ value;
  $\Gamma' := \Gamma' / \{\ell\}$;
  while there exists $T \in \tau'$ with $A(T, \ell) > 0$ **do**
    Select $T \in \tau'$ with largest $A(T, \ell)$;
    $\tau' := \tau' / \{T\}$;
    Assign T to a non-full supertask using the assignment rule
  od
od;
Remove all empty supertasks from π

Fig. 12. Heuristic algorithm for assigning tasks to supertasks.

After these tasks are assigned, the remaining objects are considered in non-increasing order by weighted contention.

Assignment rules. The act of assigning selected tasks to supertasks is delegated to an assignment rule. We consider the use of two rules, which are taken from prior work on partitioning [11,13]. The *Next Fit* (NF) rule begins by assigning tasks to the first (lowest-indexed) supertask. Whenever a supertask is unable to accept a task, the rule moves to the supertask with the next higher index and continues. Once a supertask fails to accept a task, no further attempts are made to assign to that supertask. This rule intuitively matches the goal of the heuristic due to the fact that tasks that are consecutively assigned are likely to be assigned to the same supertask.

The *First Fit* (FF) rule, on the other hand, always assigns a task to the lowest-indexed supertask that can accept the task. The advantage of this rule is that it may use fewer supertasks on average, which results in less reweighting overhead. However, the fact that supertasks are not considered in sequence, as is done under the NF rule, implies that the FF rule will be less effective at preventing object sharing across multiple supertasks. Hence, using the FF rule may reduce reweighting overhead, but is also likely to increase lock-free overhead (relative to that experienced when using the NF rule).

6.2 Implementation

Unfortunately, implementing this heuristic is a non-trivial task. The problem that arises is that the lock-free overhead depends on the assignment of tasks to supertasks. Hence, the ultimate weight of each task is not known at the time of the assignment. Effectively, each task's weight may expand or contract after being assigned to a supertask. If the total weight of a component task set after these weight changes exceeds unity, then the task assignments are not valid. In such a case, another round of assignments must be performed.

We address the dependency between task weights and supertask assignments by applying the heuristic iteratively. Specifically, an initial round of assignments is conducted using "ideal" weights[8] and an acceptance test of the form

$$\sum_{T \in \mathcal{S}} T.w \leq \alpha,$$

where α is set to unity. Once assignments are made, the weights of the component tasks are updated and each supertask is checked to ensure that the cumulative weight does not exceed unity. If this requirement is satisfied, then the assignment phase ends and the supertasks are reweighted. Otherwise, the supertask assignments are nullified, α is decremented by 0.01, and another round of assignments are conducted using the task weights computed at the end of the previous round.

The benefit of decreasing α (and hence increasingly underestimating the maximum capacity of supertasks) each time a new round begins is that a fraction $1 - \alpha$ of each supertask's maximum capacity is effectively reserved to account for the inflation of task weights. The disadvantage of such compensation is that lowering α can result in an unnecessarily high number of supertasks, thereby increasing reweighting overhead. For this reason, we use a relatively small step size (*i.e.*, 0.01) when decreasing α. We found that termination typically occurs after only two or three rounds when using this step size.

7 Experimental Results

In this section, we present the results of an experimental study that measured the lock-free overhead produced by randomly generated task sets as an upper limit, denoted B, imposed on $A(T, \ell)$ values was systematically increased. The goal of this study was to determine both the relative overhead of using lock-free objects in different ways and how this overhead scales with increasing object contention. Lock-free synchronization, both with and without supertasks, was considered.

For the supertasking cases, two different approaches to internal scheduling, called QB-EPDF and QB-EDF, were considered. Informally, QB-EPDF requires that all *subtask* deadlines be met, while QB-EDF considers only *job*

[8] Ideal weights are based on the assumption that no retries occur.

deadlines. Stated differently, QB-EPDF requires component tasks to execute at a steady rate by imposing intermediate deadlines within jobs, while QB-EDF does not. When assigning supertask weights via our reweighting approach [16,17,20], weights are assigned in proportion to the maximum amount of processor time that can be consumed by all requests for processor time (*i.e.*, either subtasks or jobs depending on which form of supertasking is used) over time intervals of a given length. Due to the intermediate subtask deadlines needed to ensure a steady rate of execution, the QB-EPDF approach typically requires higher supertask weights and hence introduces more overhead. (A more detailed discussion of supertasking and these alternatives can be found in [16].)

For each supertasking approach, each of the NF and FF rules, given in the previous section, was considered. Since we are unable to reasonably estimate the impact of algorithmic improvements obtained through the use of supertasks (like those demonstrated by the case study in Section 5), such improvements were not considered. (Improvements due to lower retry costs were, of course, considered.) The study presented here is only part of a larger study; full details can be found in [16].

Sampling. As stated, this study was based on randomly generating task sets from a sample space. We begin by defining this sample space. Four experiments was performed, each with a different processor count; the processor count (M) was assigned each of 2, 4, 8, and 16. For each experiment, the task set parameters were selected as follows:

- B was set to each value in the range $1, \ldots, 8$;
- the task count was varied across the range $6 \cdot \log_2 M, \ldots, 15 \cdot \log_2 M$;
- the total base [9] utilization was varied across the range $0.2 \cdot M, \ldots, 0.6 \cdot M$;
- the object count was varied across the range $\log_2 M, \ldots, 5 \cdot \log_2 M$;
- each $A(T, \ell)$ value was selected from the range $1, \ldots, B$.

These ranges were chosen somewhat arbitrarily so that the range of values that we expect to observe in practice would be represented. Some parameters were systematically varied, as mentioned above, to ensure a reasonably complete coverage of the sample space.

Measurement. The following measurements were taken during the experimental runs.

Ideal: the sum of the ideal weights [10] of all tasks (a baseline measurement);

[9] Lock-free overhead was not considered.

[10] A task's ideal weight is based on the assumption that no retries are needed for object accesses.

this measurement reflects mapping overhead[11] alone.

No Supertasks: the total weight of all tasks when object accesses are considered and supertasks are not used; this measurement reflects mapping and retry overhead.

Ideal QB (FF): the sum of the ideal weights of all supertasks created by the FF assignment heuristic (a baseline measurement); this measurement reflects mapping and retry overhead.

Ideal QB (NF): the sum of the ideal weights of all supertasks created by the NF assignment heuristic (a baseline measurement); this measurement reflects mapping and retry overhead.

QB-EPDF (FF): the sum of the assigned weights of all supertasks created by the FF assignment heuristic under the QB-EPDF approach; this measurement reflects mapping, retry, and reweighting overhead.

QB-EPDF (NF): the sum of the assigned weights of all supertasks created by the NF assignment heuristic under the QB-EPDF approach; this measurement reflects mapping, retry, and reweighting overhead.

QB-EDF (FF): the sum of the assigned weights of all supertasks created by the FF assignment heuristic under the QB-EDF approach; this measurement reflects mapping, retry, and reweighting overhead.

QB-EDF (NF): the sum of the assigned weights of all supertasks created by the NF assignment heuristic under the QB-EDF approach; this measurement reflects mapping, retry, and reweighting overhead.

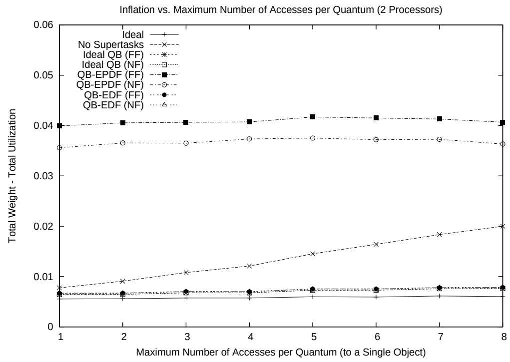
When discussing the supertasking approaches, apparent trends in the reweighting overhead will not be noted. These trends can be distinguished from the lock-free trends by comparing the "Ideal QB" measurements with the QB-EPDF and QB-EDF measurements.

Results. Figure 13 shows the weight inflation plotted against B. A 99% confidence interval was computed for each sample mean; these intervals are shown in Figure 14. As shown, overhead can be significantly reduced through the use of supertasks. However, the degree of improvement varies depending on the approach. For the remainder of the section, we will discuss some of the specific relationships shown in Figure 13.

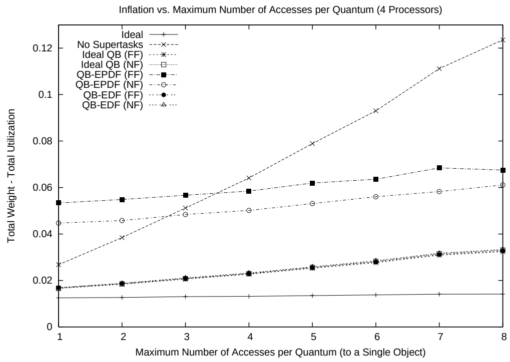
First, notice that the "Ideal QB (FF)" and "QB-EDF (FF)" lines (respectively, the "Ideal QB (NF)" and "QB-EDF (NF)" lines) are virtually co-linear. This suggests little to no reweighting overhead is suffered under the QB-EDF approach on average. As shown, this is not the case under the QB-EPDF approach.

Second, notice that the impact of B in the supertasking cases is negligible when $M = 2$. On two processors, typically only one or two supertasks are

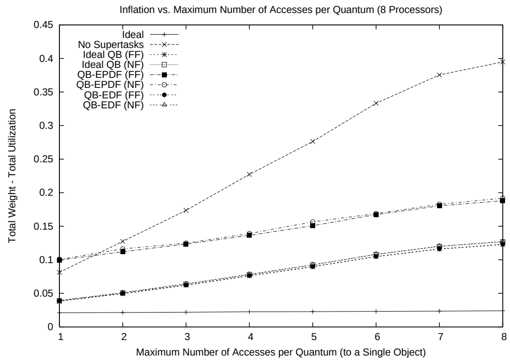
[11] Mapping overhead results when assigning a weight based on a mapping rule.



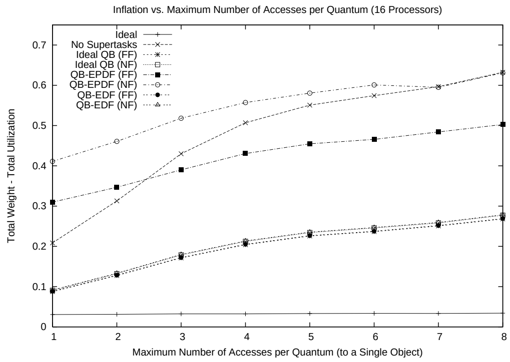
(a) $M = 2$



(b) $M = 4$



(c) $M = 8$



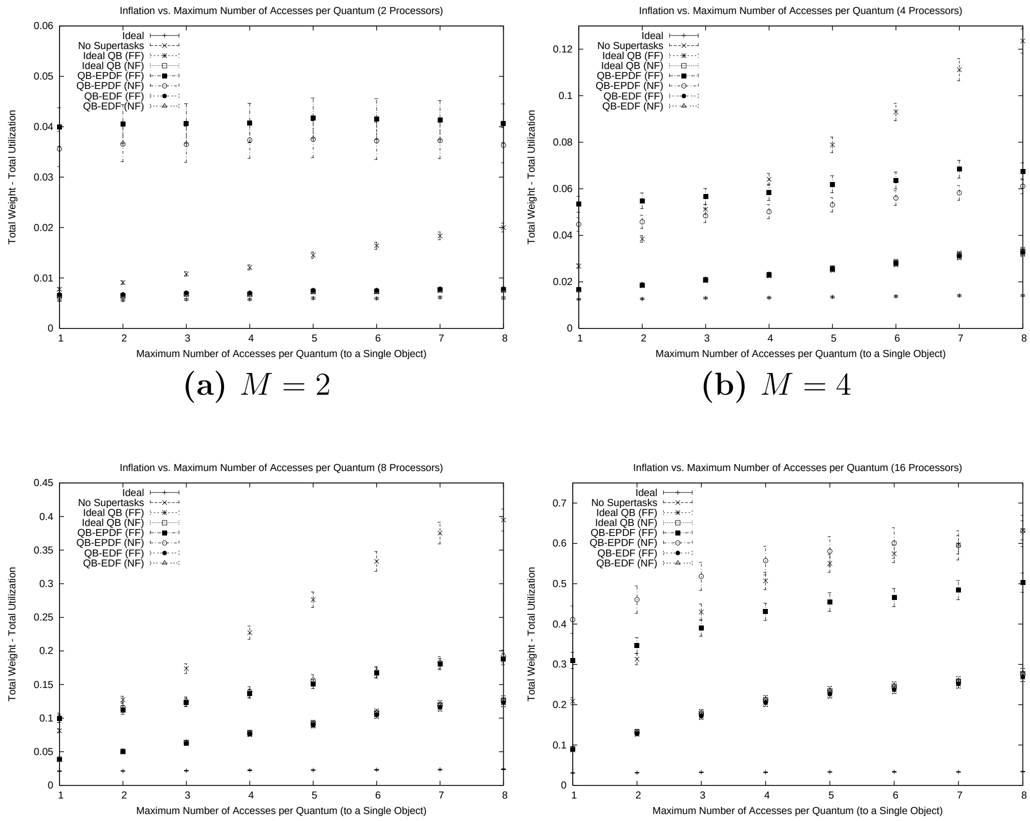
(d) $M = 16$

Fig. 13. Plots show how inflation varies as the bound on the per-quantum number of object accesses is increased when using lock-free synchronization on two processors. The figure shows the sample means for the **(a)** $M = 2$, **(b)** $M = 4$, **(c)** $M = 8$, and **(d)** $M = 16$ cases.

needed. Assigning tasks in any reasonable fashion results in little to no sharing between supertasks. As a result, retry overhead tends to be insignificant.

Third, the reweighting overhead experienced under QB-EPDF tends to outweigh the reduction in retry overhead when $B < 3$. As explained earlier, QB-EPDF requires that all subtask deadlines be met. Hence, it experiences more reweighting overhead than the QB-EDF approach. The benefit of using supertasks is determined by the relative magnitudes of the retry overhead and the reweighting overhead introduced by the supertasks. For small B, retry overhead is fairly low. As a result, the reweighting overhead introduced by QB-EPDF exceeds the retry overhead avoided by the use of supertasks.

Finally, the NF rule appears to produce more overhead on average than the FF rule. Because the NF rule abandons a supertask on the first assignment failure, it tends to create more supertasks than the FF rule. To understand why this leads to more overhead, suppose that each component task set has a total utilization of 0.92 (on average) under the FF rule, but a total utilization



(a) $M = 2$ **(b)** $M = 4$

(c) $M = 8$ **(d)** $M = 16$

Fig. 14. Plots show how inflation varies as the bound on the per-quantum number of object accesses is increased when using lock-free synchronization on two processors. The figure shows the 99% confidence intervals for the **(a)** $M = 2$, **(b)** $M = 4$, **(c)** $M = 8$, and **(d)** $M = 16$ cases.

of only 0.8 under the NF rule. When the total task set utilization is 16, the FF rule can be expected to create around $\lceil 16/0.92 \rceil = 18$ supertasks. On the other hand, the NF rule should produce around $\lceil 16/0.8 \rceil = 20$ supertasks. Because, the NF rule tends to produce more supertasks on average, it also tends to incur more reweighting overhead.

8 Conclusion

In this paper, we have addressed the problem of synchronizing access to simple shared objects in Pfair-scheduled multiprocessor systems by considering the use of lock-free techniques. We have presented analysis to support the use of lock-free objects under Pfair scheduling. We have also shown that lock-free overheads can be reduced by restricting parallelism through the use of supertasks. In addition, we presented a simple heuristic for assigning tasks to supertasks and experimentally evaluated its effectiveness.

These contributions are significant in part because lock-free techniques are generally considered impractical on real-time multiprocessors. This view is based on our inability to bound the worst-case number of interferences that occur across processors during an operation. However, as we have shown, the structure provided by quantum-based scheduling enables the efficient use of lock-free techniques under Pfair scheduling and other similar approaches.

Related topics, such as supertasking and locking synchronization, are discussed in detail in [16].

References

[1] 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 Symposium*, pages 92–105. December 1996.

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

[3] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 35–43, June 2000.

[4] J. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *Proceedings of the Seventh International Conference on Real-time Computing Systems and Applications*, pages 297–306, December 2000.

[5] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-time Systems*, pages 76–85, June 2001.

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

[7] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[8] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[9] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 161–170. December 2001.

[10] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: a proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, October 2000.

[11] S. Davari and S. Dhall. An on-line algorithm for real-time tasks allocation. In *Proceedings of the 7th IEEE Real-time Systems Symposium*, pages 194–200. December 1986.

[12] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 171–180. December 2001.

[13] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[14] P. Gai, G. Lipari, and M. di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 73–83. December 2001.

[15] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Bell Telephone Laboratories, Inc. 1979.

[16] P. Holman. On the implementation of Pfair-scheduled multiprocessor systems. PhD Thesis, University of North Carolina at Chapel Hill. 2004.

[17] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, December 2001.

[18] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 111-120, June 2002.

[19] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 149–158, December 2002.

[20] P. Holman and J. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 41–50, July 2003.

[21] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 130–140, August 1994.

[22] G. Lamastra, G. Lipari, and Luca Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 151–160. December 2001.

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

[24] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the Twentieth IEEE Real-time Systems Symposium*, pages 294–303, December 1999.

[25] A. Mok. Fundamental design problems for the hard real-time environment. Ph.D. Thesis, Massachussetts Institute of Technology. 1983.

[26] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[27] R. Rajkumar. *Synchronization in Real-time systems – A priority inheritance approach*. Kluwer Academic Publishers, Boston, 1991.

[28] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Ninth IEEE Real-time Systems Symposium*, pages 259–269. 1988.

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

[30] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 51–59, June 2003.