Object Sharing in Pfair-scheduled Multiprocessor Systems*

Philip Holman and James H. Anderson

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
Phone: (919) 962-1757
Fax: (919) 962-1799
E-mail: {holman, anderson}@cs.unc.edu

December 2001

Abstract

We consider various techniques for implementing shared objects and for accounting for object-sharing overheads in Pfair-scheduled multiprocessor systems. We primarily focus on the use of lock-free objects, though some lock-based alternatives are briefly considered as well. Lock-free objects are more economical for implementing relatively simple objects such as buffers, stacks, queues, and lists; locking techniques are preferable for more complicated objects and for sychronizing accesses to physical devices. We present schedulability conditions for Pfair-scheduled systems in which lock-free objects are used. In addition, using shared queues as an example, we show how one can exploit the tight synchrony that exists in Pfair-scheduled systems to optimize lock-free implementations. We also show that lock-free object-sharing overheads can be reduced by combining tasks into supertasks; this is because, *within* a supertask, less-costly uniprocessor synchronization techniques can be used.

*Work supported by NSF grants CCR 9972211, CCR 9988327, and ITR 0082866.

1 Introduction

There has been much recent work on scheduling techniques that ensure fairness, temporal isolation, and timeliness among tasks multiplexed on the same 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, where feasible (timeliness).

In recent years, there has been considerable interest in fair scheduling algorithms for multiprocessor systems [3, 4, 5, 7, 8, 9, 16]. 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 [9].

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, 18, 19, 20, 21]. 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 [10, 11, 14, 16].)

In this paper, we consider the problem of object sharing in fair-scheduled multiprocessor systems. We primarily focus on the use of lock-free objects, which are implemented without critical sections or related mechanisms. However, some lock-based alternatives are briefly considered as well. We take as our notion of fairness the *Pfairness* constraint proposed by Baruah *et al.* [7]. We also limit attention to *periodic* task systems [15]. Although we consider only Pfair-scheduled periodic tasks, many of our results are applicable to other fair scheduling algorithms and notions of recurrent execution as well.

To reduce object-sharing overheads, we consider the use of supertasking, which was proposed previously to reduce migration costs in Pfair-scheduled systems [17]. A *supertask* is merely a collection of periodic tasks that is scheduled as a single entity; when a supertask is scheduled, it selects one of its component tasks for execution. We show that, by combining tasks into supertasks, synchronization overheads can be reduced. This is because a supertask executes its component tasks as a virtual uniprocessor; hence, contention for objects is reduced, and less-costly uniprocessor synchronization techniques often can be used.

The rest of this paper is organized as follows. In Sec. 2, we consider Pfair scheduling and supertasking in greater detail. Then, in Sec. 3, we consider several problems that occur when synchronizing tasks in Pfair-scheduled systems. In Sec. 4, we present schedulability conditions for Pfair-scheduled systems comprised of periodic tasks that share lock-free objects, with and without supertasking. In Sec. 5, two implementations of a shared queue are given to demonstrate the algorithmic benefits of supertasking. We then present and experimentally evaluate a heuristic for assigning tasks to supertasks in Sec. 6. We conclude in Sec. 7.

2 Pfair Scheduling and Supertasking

Let τ denote a set of *periodic* tasks to be scheduled on $M \geq 2$ processors. Let ρ denote the set of objects shared by tasks in τ. Assume that each object $\ell \in \rho$ is accessed by at least two tasks and that each task accesses at least one object.

Pfair scheduling. Under Pfair scheduling, each task has a *weight*, which determines the manner in which it is scheduled. Let $T.w$ denote the weight of task $T \in \tau$. $T.w$ is determined (as explained later) by considering two parameters associated with T: an integer *period* $T.p$, and a *base execution cost* $T.e$, which is defined as the worst-case cost of a single job (*i.e.*, invocation) of T *without considering shared-object accesses*. Unlike prior work on Pfair scheduling, we do *not* assume that $T.e$ is an integer here.

Pfair scheduling algorithms allocate processor time in discrete time units, or quanta. The interval $[t, t+1)$, where t is a non-negative integer, is called *time slot* t. In each time slot, each processor can be assigned to at most one task and each task may be assigned at most one processor. Task migration is allowed. The sequence of scheduling decisions made in these time slots defines a *schedule*. A schedule respects Pfairness if and only if the following property holds for all $T \in \tau$ and for all $t \geq 0$ [12].

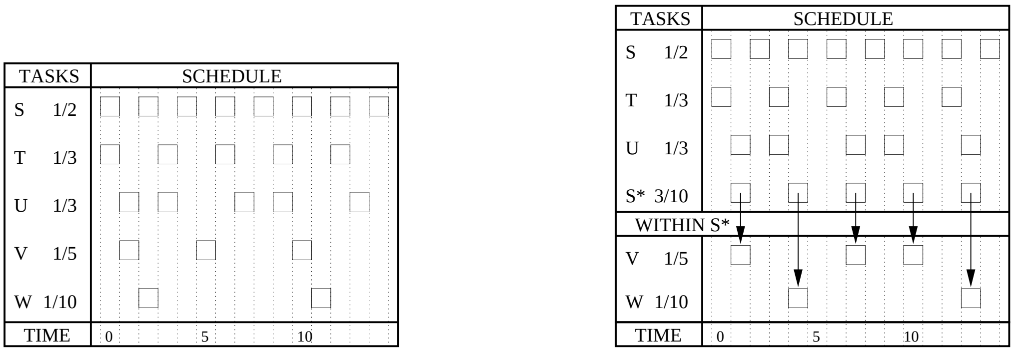
PF: T receives either $\lfloor T.w \cdot t \rfloor$ or $\lceil T.w \cdot t \rceil$ quanta over any interval $[0, t)$.

At present, three optimal Pfair scheduling algorithms are known: PF [7], PD[8], and PD^2 [3, 4, 5]. Because these algorithms are optimal, each will produce a Pfair schedule whenever some such schedule exists. Baruah *et al.* [7] derived the following necessary and sufficient condition for the existence of a Pfair schedule for a task set τ on M processors.

$$\sum_{T \in \tau} T.w \leq M \tag{1}$$

Baruah *et al.* also demonstrated that a rational task weight $T.w = \frac{b}{c}$ ensures that *exactly* b quanta are received in the interval $[k \cdot c, (k+1) \cdot c)$ for all $k \geq 0$ in *any* schedule that respects Pfairness. Therefore, our goal in this paper is to determine a worst-case object-sharing overhead $T.\Delta$ for each task T such that the worst-case per-job execution cost of task T is upper-bounded by $\lceil T.e + T.\Delta \rceil$. In this case, the weight assignment $T.w = \frac{\lceil T.e + T.\Delta \rceil}{T.p}$ will be sufficient to ensure that all job deadlines of T are met in any Pfair schedule. Notice that the ceiling is necessary to ensure that the weight is rational.

Example. To illustrate Pfair scheduling, consider the schedule shown in Fig. 1(a), which was produced by the PD^2 scheduler. In this schedule, two processors are shared among five tasks, labeled S–W, which are assigned weights 1/2, 1/3, 1/3, 1/5, and 1/10, respectively. Vertical dashed lines mark the quantum boundaries and boxes show when each task is allocated a processor. Notice the even distribution of each task's allocation. This distribution of allocation both simplifies multiprocessor scheduling and complicates object sharing, as will be explained later in Sec. 3.



(a) PD^2 schedule **(b) Supertasking schedule**

Figure 1: Sample Pfair schedules for a task set consisting of five tasks with weights 1/2, 1/3, 1/3, 1/5, and 1/10 respectively. **(a)** Normal schedule produced when no supertasks are used. **(b)** Schedule produced when tasks V and W are combined into the supertask S^*, which competes with weight 3/10.

Supertasking. In *supertasking* [17, 12], the task set τ is partitioned into a collection of non-empty subsets, σ. Each $\mathcal{S} \in \sigma$, 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.) We let $\mathcal{S}(T)$ denote the supertask of which task T is a component task.

To illustrate supertasking, consider Fig. 1(b). The schedule shown is derived from that of Fig. 1(a) by letting $\sigma = \{\{S\}, \{T\}, \{U\}, \{V, W\}\}$. Here, S^* is a supertask with component tasks V and W and competes with weight $S^*.w = 3/10 = V.w + W.w$. When S^* is scheduled in the upper schedule, it selects one of its component tasks to execute (shown in the lower schedule) as indicated by the arrows. (Note that, although S^* executes V and W as a virtual uniprocessor, these tasks actually may migrate.)

Although in this example S^*'s weight is equal to the cumulative weight of its component tasks, Moir and Ramamurthy [17] demonstrated that such a simple weight assignment may result in deadline misses when used with PF, PD, or PD^2. In previous work, we showed that component-task deadlines can be guaranteed by *reweighting* supertasks so that they use a slightly larger weight [12]. It is unknown whether supertasking without weight inflation is inherently suboptimal. In the experimental evaluations that we present later in Sec. 6, we present results for supertasking with *and* without weight inflation for this reason.

3 Object Sharing in Fair-scheduled Systems

Fair schedulers distribute each task's allocation evenly across any interval of time. Requiring a task to execute at a steady rate is problematic when blocking synchronization is introduced. If a task is blocked due to a lock request, it is incapable of executing. Even worse, a lock-holding task cannot always execute nonpreemptively to reduce blocking times since such behavior might violate fairness. For these reasons, techniques that avoid blocking are preferable in fair-scheduled systems, where applicable.

```
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)
```

Figure 2: Lock-free enqueue implementation.

Lock-free algorithms. Lock-free algorithms work particularly well for simple objects like buffers, queues, and lists. In such algorithms, object calls are implemented using "retry loops." Fig. 2 depicts a lock-free enqueue operation that is implemented in this way. An item is enqueued in this implementation by using a *two-word compare-and-swap* (`CAS2`) instruction[1] to atomically update a tail pointer and either the "next" pointer of the last item in the queue or a head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the `CAS2` instruction succeeds. An important property of lock-free implementations such as this is that operations may *interfere* with each other. An interference results in this example when a successful `CAS2` by one task causes another task's `CAS2` to fail.

In addition to lock-free algorithms, we will also consider "wait-free" algorithms in this paper. The wait-free requirement strengthens the lock-free requirement by guaranteeing that each operation will eventually make progress. For this reason, wait-free algorithms may consist only of code segments with a bounded number of loop iterations.

When lock-free objects are used in real-time systems, bounds on loop retries must be computed for scheduling analysis. On uniprocessors, aspects of priority schedulers can be taken into account to determine such bounds [2]. In real-time multiprocessor systems, lock-free algorithms have been viewed as being impractical, because deducing bounds on retries due to interferences *across* processors is difficult. However, as we later show, the tight synchrony in Pfair-scheduled systems can be exploited to help bound interferences more tightly than in an asynchronous multiprocessor system. In addition, the use of supertasking in Pfair-scheduled systems may permit the use of uniprocessor wait-free object implementations in place of multiprocessor lock-free variants. For most objects, these wait-free implementations will be more efficient. (We illustrate this fact with an example in Sec. 5.)

Lock-based algorithms. Despite the expected superiority of lock-free techniques, they cannot be applied in all cases. The effectiveness of lock-free algorithms is limited to operations that can be completed quickly. For some lengthy operations, lock-based techniques may be a necessity. Hence, lock-based tech-

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

niques cannot be totally ignored in fair-scheduled systems. We will now briefly explain how some existing lock-based concepts can be applied in fair-scheduled systems. We intend to examine these lock-based techniques in more detail in future work.

In real-time systems in which locks are used, *priority inversions* must be dealt with. A priority inversion occurs when a task is blocked by a task of lower priority. Inheritance and ceiling schemes [6, 18, 19, 20, 21] limit the duration of priority inversions by temporarily boosting a lock-holding task's priority when it blocks any higher-priority task. In fair-scheduled systems, such schemes cannot be applied directly because they disrupt allocation rates. As a result, blocking times in fair-scheduled systems are likely to be much worse than in systems that do not require fair scheduling.

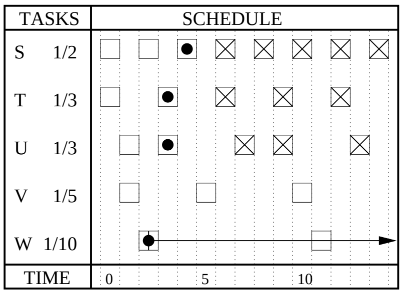
As an example, reconsider the schedule shown in Fig. 1(a), which shows the PD^2 schedule for five *independent* tasks on two processors. Now consider Fig. 3(a), in which the tasks are no longer independent. In time slot 2, suppose task W obtains a lock, which it will not release for two quanta. When S, T, and U request this lock in time slots 3-4, they are forced to wait until the lock is released by W. Due to W's low weight, this does not happen until sometime after time slot 14. In the meantime, all quanta allocated to S, T, and U are useless (because they are blocked).

Although conventional inheritance and ceiling protocols are not directly applicable in fair-scheduled systems, many of the concepts underlying them are applicable. Disruptions to fair allocation rates can be ameliorated by decoupling the choice of which task to schedule in a particular time slot from the choice of which task to *charge* for the execution time. The following list describes several alternatives based on this idea that we intend to explore in future work.

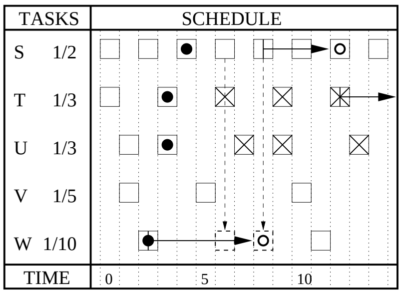
Rate inheritance. This approach is based on the priority inheritance protocol [21]. Since allocation rates are proportional to weights in Pfair-scheduled systems, we can speed a lock-holding task's critical-section execution by letting it execute within any quantum allocated to the heaviest task that it blocks (if it indeed blocks a heavier task). Fig. 3(b) illustrates this technique. In time slot 3, W inherits T's scheduling parameters, and in time slot 4, it inherits S's parameters. W executes within the quanta allocated to S in time slots 6 and 8.

Allocation inheritance. In this approach, a lock-holding task inherits all quanta allocated to *all* tasks that it blocks. This is similar to the concept of bandwidth inheritance, discussed in [14]. The effectiveness of this approach in multiprocessor systems is limited by the possibility that many of the inherited quanta may be allocated in parallel. As an example, consider Fig. 3(c). Notice that W receives allocations from both S in time slot 6 and U in time slot 7. Also, notice that two blocked tasks are scheduled in time slot 6, hence one quantum is wasted.

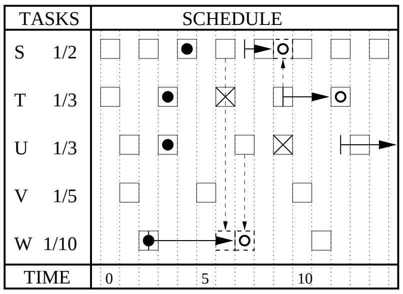
Weight inheritance. The problem of parallel quantum allocations can be avoided by adding to a lock-holding task's weight the weight of each task that it blocks, rather than by simply inheriting their quanta.



(a) Simple locking



(b) Rate inheritance



(c) Allocation inheritance

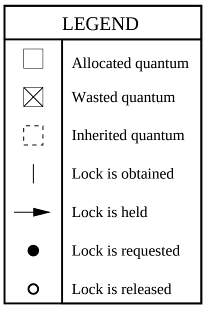


Figure 3: A series of two-processor Pfair schedules for an non-independent task set consisting of five tasks with weights 1/2, 1/3, 1/3, 1/5, and 1/10 respectively. **(a)** Simple locking **(b)** Rate inheritance **(c)** Allocation inheritance

In this manner, weight inheritance is also derived from the concept of bandwidth inheritance. By inheriting the weight of all blocked tasks, the lock-holding task can guarantee that the number of quanta it receives is the highest possible with respect to the available bandwidth. However, at present, Pfair scheduling only permits statically-defined task weights, which prevents the use of such an approach.

Though these and similar techniques may be potentially useful in fair-scheduled systems, it is doubtful that they will outperform lock-free techniques when both are applicable. For this reason, we focus on lock-free techniques in the rest of this paper. However, a paper on lock-based techniques will be forthcoming.

4 Schedulability Conditions

We now state and prove sufficient schedulability conditions for a system of Pfair-scheduled tasks that share lock-free objects. To the best of our knowledge, this is the first work involving the use of lock-free objects in real-time multiprocessors. The efficient use of lock-free algorithms is made possible by the tight synchrony provided by Pfair scheduling. In addition, supertasks can be used to reduce the number of retries caused by concurrent object accesses. Indeed, if all tasks accessing an object can be packed into a single supertask, then a uniprocessor implementation of the object may be used instead of the multiprocessor implementation (as explained later). We demonstrate the benefits of such a substitution later in Sec. 5.

Definitions and assumptions. Assume that each object $\ell \in \rho$ is accessed at most $J_T(\ell)$ (respectively, $Q_T(\ell)$) times within a single job (quantum) of task T. Further, assume that each access to ℓ has a base cost of $b(\ell)$ and a retry cost of $r(\ell)$. Let $B^{[M]}(\ell)$ and $R^{[M]}(\ell)$ denote the the base and retry costs, respectively, for an M-processor implementation of ℓ. Then, $B^{[1]}(\ell)$ and $R^{[1]}(\ell)$ denote the costs associated with a uniprocessor implementation. As we shall see, $b(\ell)$ will be defined to be either $B^{[M]}(\ell)$ or $B^{[1]}(\ell)$ depending on how tasks are assigned to supertasks (and similarly for $r(\ell)$). Note that we are assuming that the costs of all operations of ℓ are the same; this assumption could be eliminated, at the price of slightly more complicated notation and analysis.

For systems with supertasks, we let $Q_{\mathcal{S}}(\ell) = \mathsf{max}\{Q_T(\ell) \mid T \in \mathcal{S}\}$, which is the worst-case number of accesses to $\ell \in \rho$ in a single quantum by *any* component task of $\mathcal{S}$. We will use $\mathcal{S}.w$ to denote the minimum weight *at which the safety of $\mathcal{S}$'s component tasks can be guaranteed*. (As mentioned earlier, supertask weights may need to be inflated to guarantee component-task safety.)

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 some object access. This bounds the number of retries to at most the number of concurrent accesses to an object and prevents two tasks from livelocking due to repeated mutual interference.
- **Preemption Assumption (PA):** A single object access will be preempted (*i.e.*, cross a quantum boundary) at most once. This assumption comes from previous work [1] on lock-free algorithms and is based on the observation that lock-free operations typically are very short in comparison to the length of a quantum.

We also define a few shorthand notations. Let $\mathsf{maxsum}_k\{a_1, \ldots, a_n\}$ be the maximum value produced by summing $\mathsf{min}(k,n)$ elements from the multiset $\{a_1, \ldots, a_n\}$. For example, $\mathsf{maxsum}_2\{2, 2, 1\} = \mathsf{max}\{2 + 2, 2{+}1, 2{+}1\} = 4$, $\mathsf{maxsum}_4\{2, 2, 1\} = \mathsf{max}\{2{+}2{+}1\} = 5$, and $\mathsf{maxsum}_0\{2, 2, 1\} = \mathsf{max}\{0\} = 0$. In addition, let $I_T(\ell) = \mathsf{maxsum}_{M-1}\{Q_U(\ell) \mid U \in \tau - \{T\}\}$ and let $I_{\mathcal{S}}(\ell) = \mathsf{maxsum}_{M-1}\{Q_{\mathcal{T}}(\ell) \mid \mathcal{T} \in \sigma - \{\mathcal{S}\}\}$. (Recall that τ is the set of tasks and σ is the set of supertasks.)

4.1 Feasibility Without Supertasks

Since at least two tasks are assumed to share each object $\ell \in \rho$, we let $b(\ell) = B^{[M]}(\ell)$ and $r(\ell) = R^{[M]}(\ell)$ in the analysis that follows.

Theorem 1 *If task T competes with weight*

$$T.w = \frac{\left\lceil T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot [b(\ell) + (2I_T(\ell) + 1) \cdot r(\ell)] \right\rceil}{T.p},$$

where $T.w \leq 1$, then each job of T will complete by its deadline in any schedule that respects Pfairness.

Proof: By (PF), a Pfair schedule guarantees that each task T with a rational weight $T.w$ equal to the expression given in the theorem will receive $\left\lceil T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot [b(\ell) + (2I_T(\ell) + 1) \cdot b(\ell)] \right\rceil$ quanta during the interval $[k \cdot T.p,\ (k+1) \cdot T.p)$ for all $k \geq 0$. Consider a single job of T and a single access to some object $\ell \in \rho$. 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 the quantum. Since there are at most $M-1$ tasks executing in parallel with T and each concurrent task U makes at most $Q_U(\ell)$ 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 a single quantum. Therefore, at most $2I_T(\ell) + 1$ 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 while accessing ℓ, then one additional retry may be needed.) Therefore, $b(\ell) + (2I_T(\ell) + 1) \cdot r(\ell)$ is an upper bound on the worst-case execution cost of one access to ℓ. Since at most $J_T(\ell)$ accesses to ℓ occur in one job, the per-job execution requirement of T can be no more than $T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot [b(\ell) + (2I_T(\ell) + 1) \cdot b(\ell)]$, which is at most $\left\lceil T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot [b(\ell) + (2I_T(\ell) + 1) \cdot b(\ell)] \right\rceil$. Therefore, T's job deadlines will be met. $\square$

Corollary 1 *If for all $T \in \tau$, $T.w = (\lceil T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot [b(\ell) + (2I_T(\ell) + 1) \cdot r(\ell)] \rceil / T.p) \leq 1$, and if $\sum_{T \in \tau} T.w \leq M$, then τ is feasible on M processors.*

4.2 Feasibility With Supertasks

Supertasking can improve performance in two ways.

- A supertask can prevent a collection of potentially-interfering tasks from executing in parallel. By doing so, the worst-case number of retries needed to complete an object access can be reduced.
- By (PA), if all tasks that access an object can be placed in the same supertask, then at most one retry is needed during any access. (A retry is necessary only if the object access is preempted.) Since the number of retries is constant in this case, the lock-free algorithm becomes a wait-free algorithm and can usually be simplified considerably.

When supertasks are used, the worst-case number of interferences experienced in a single quantum changes from $I_T(\ell)$ to $I_{\mathcal{S}}(\ell)$. An object's implementation can be selected based simply upon this value. Notice that the definition of $I_{\mathcal{S}}(\ell)$ implies that its value is zero only when all tasks that share ℓ are component tasks of $\mathcal{S}$. From this, $b(\ell)$ and $r(\ell)$ can be determined easily using the following definition.

$$b(\ell),\ r(\ell) = \begin{cases} B^{[1]}(\ell),\ R^{[1]}(\ell) & \text{if } I_{\mathcal{S}}(\ell) = 0 \text{ for some } \mathcal{S} \\ B^{[M]}(\ell),\ R^{[M]}(\ell) & \text{otherwise} \end{cases}$$

Task	e	p	J_1	Q_1	J_2	Q_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)

Object	$B^{[1]}$	$R^{[1]}$	$B^{[M]}$	$R^{[M]}$
ℓ_1	0.012	0.08	0.05	0.16
ℓ_2	0.005	0.075	0.017	0.11

(b)

Task	I_1	I_2	δ_1	δ_2	Δ_1	Δ_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)

Figure 4: **(a)** A sample task set τ on $M = 4$ processors with two lock-free objects ℓ_1 and ℓ_2. **(b)** Parameters of ℓ_1 and ℓ_2. **(c)** Summary of all values computed when using Corollary 1 to assign weights to tasks.

Theorem 2 *If component task T of a safely-weighted supertask competes with weight*

$$T.w = \frac{\left\lceil T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot \left[b(\ell) + (2I_{\mathcal{S}(T)}(\ell) + 1) \cdot r(\ell) \right] \right\rceil}{T.p},$$

where $T.w \leq 1$, then each job of T will complete by its deadline in any schedule that respects Pfairness.

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

Corollary 2 *If for all $T \in \tau$, $T.w = (\lceil T.e + \sum_{\ell \in \rho} J_T(\ell) \cdot \left[b(\ell) + (2I_{\mathcal{S}(T)}(\ell) + 1) \cdot r(\ell) \right] \rceil / T.p) \leq 1$, and if $\mathcal{S}.w \leq 1$ for each supertask $\mathcal{S} \in \sigma$, and if $\sum_{\mathcal{S} \in \sigma} \mathcal{S}.w \leq M$, then τ is feasible on M processors.*

4.3 Examples

To demonstrate the use of these results, we will compute Pfair scheduling weights for the simple task set shown in Fig. 4(a)–(b) on a four-processor system.

Scheduling without supertasks. Fig. 4(c) summarizes the values that are computed when assigning weights to tasks based on Corollary 1. We will explain each column in turn by considering 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 $I_T(\ell)$ yields $I_{T_{10}}(\ell_1) = \mathsf{maxsum}_{M-1}\{Q_U(\ell_1) \mid U \in \tau - \{T_{10}\}\} = \mathsf{maxsum}_3\{2, 1, 1, 1, 1, 0, 0, 0, 0\} = 4$. This is shown in the column labeled I_1 in Fig. 4(c). Next, we compute the worst-case execution cost of a single access to object ℓ_1 by T_{10}, labeled δ_1 in the table. By previous observations, we know that $b(\ell_1) = B_1^{[M]} = 0.05$ and $r(\ell_1) = R_1^{[M]} = 0.16$. From these values and

Supertask	Components	Q_1	Q_2	I_1	I_2
$\mathcal{S}_1$	T_1, T_2, T_6, T_8	2	0	2	3
$\mathcal{S}_2$	T_3, T_4, T_5, T_7, T_9, T_{10}	2	3	2	0

(a)

Task	I_1	I_2	δ_1	δ_2	Δ_1	Δ_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)

Figure 5: **(a)** Parameters for the partitioning $\sigma = \{\{T_1, T_2, T_6, T_8\}, \{T_3, T_4, T_5, T_7, T_9, T_{10}\}\}$. **(b)** Summary of all values computed when using Corollary 2 to assign weights to tasks.

the expression $\delta_T(\ell) = b(\ell) + (2I_T(\ell) + 1) \cdot r(\ell)$, we get $\delta_{T_{10}}(\ell_1) = 0.05 + (2 \cdot 4 + 1) \cdot 0.16 = 1.49$. We can now determine the total execution overhead of T_{10}'s accesses to ℓ_1 in a single job, labeled Δ_1 in the table, by simply multiplying δ_1 by $J_{T_{10}}(\ell_1)$. Doing so yields $\Delta_{T_{10}}(\ell_1) = 5 \cdot 1.49 = 7.45$. In a similar manner, it can be shown that $\Delta_{T_{10}}(\ell_2) = 12.084$. By Corollary 1, T_{10} should be assigned the weight $T_{10}.w = \frac{\lceil 50+7.45+12.084 \rceil}{700} = \frac{\lceil 69.534 \rceil}{700} = \frac{70}{700}$, as shown in the last column of the table. Note that feasibility is not guaranteed unless both conditions in Corollary 1 hold. However, each weight in the last column of Fig. 4(c) is easily seen to be less than one, and $\sum_{T \in \tau} T.w \approx 1.57 \leq 4$.

Scheduling with supertasks. Fig. 5(a) shows one possible partitioning of the example task set from Fig. 4(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, $b(\ell_2) = B_2^{[1]} = 0.005$ and $r(\ell_2) = R_2^{[1]} = 0.075$. However, notice that the multiprocessor version of ℓ_1's algorithm must still be used. Hence, $b(\ell_1) = B_1^{[M]} = 0.05$ and $r(\ell_1) = R_1^{[M]} = 0.16$.

Since the computations in Corollary 2 are similar to those already demonstrated, we do not explain them here. The values resulting from each step can be seen in Fig. 5(b). Notice that many of the task weights are smaller than in the previous example. In addition, the sum of the task weights has been reduced from 1.57 to 1.45. However, the use of supertasking introduces an inflation term, due to supertask reweighting, that will likely negate this benefit for this simple example. We will not present the details of the reweighting approach since they are not directly related to the process of computing task weights. Instead, we refer the interested reader to our previous work on that subject [12].

In Sec. 6, we present experimental results that suggest that supertasking often improves schedulability, even with reweighting. In a system with relatively few processors, the benefit of supertasking is often negated by the inflation overhead. This is because of the limited parallelism in such systems. As the degree of parallelism increases, the benefits of supertasking increase while its overhead remains relatively constant. (Inflation in the reweighting approach depends on the magnitudes of the component task weights and is independent of both the number of tasks and the number of processors in the system.)

5 Case Study: Queues

In this section, we present two lock-free queue implementations that demonstrate how algorithmic simplifications may be obtained by applying the supertasking approach.

Lock-free algorithms often employ the *compare-and-swap* (`CAS`) primitive to ensure the safety of state variable updates. `CAS` is similar to the `CAS2` primitive used in Fig. 2, but accesses only a single memory location. (`CAS` or related primitives are available on most modern machines.) By using `CAS` to update state variables, and by tagging each such variable with a counter that is incremented with each update, it can be ensured that "out-of-date" updates that could corrupt the implemented object's state have no effect. This mechanism is used in the implementations presented here. To simplify the presentation of our algorithms, we use the template definition shown below to define such tagged state variables. The *tag* field holds the variable's current tag value, while the *val* field holds the variable's current logical value.

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

It is assumed that all tagged variables require only a *single word of memory* for both the *tag* and *val* fields combined. Though bounding the range of counters is important, we ignore this issue here to simplify the presentation of the algorithms, and make the simplifying assumption that the range of each tag field is unbounded. (It is actually possible to bound the size of these counters using the results of our scheduling analysis in Sec. 4. However, deriving counter bounds is outside the scope of this paper.)

Our multiprocessor and uniprocessor queue implementations are shown in Figs. 7 and 8, respectively. Both implementations use a linked list data structure, in which each list node consists of a value field, *val*, and a tagged pointer to the next list node, *next*. In addition, the tagged state variables *Head* and *Tail* record the start and end, respectively, of the list. To allow safe concurrent enqueue and dequeue operations, the linked list has a *dummy* head node. The structure of the queue is illustrated in Fig. 6.

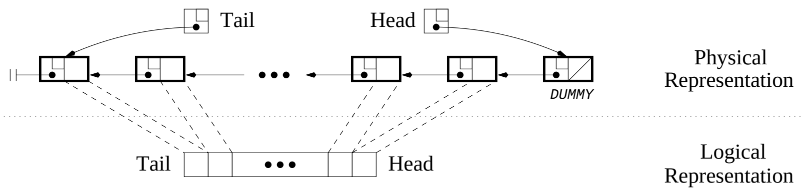


Figure 6: Physical and logical representation of the shared queue.

The *next* fields and the *Tail* variable are tagged to help synchronize multiple enqueuers (writers). For the single-enqueuer case, the enqueue procedure simplifies to `Enqueue-1W`, shown below `Enqueue-MW` in Fig. 7, and these tag fields may be removed. Similarly, the tag field in *Head* synchronizes multiple dequeuers (readers) and can be removed in the single-dequeuer case. For this case, `Dequeue-MR` simplifies to the procedure `Dequeue-1R` in Fig. 7. `Enqueue-1W` and `Dequeue-1R` are actually common to both implementations, and thus are not repeated in Fig. 8.

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

typedef node:
    record
        val: eltype;
        next: tagged(pointer to node)

shared var
    Head, Tail: tagged(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.val).next;
6:      if t = Tail then
7:          done := CAS(&(*(t.val).next),
                    (x.tag,nil),(x.tag+1,in));
8:          x := *(t.val).next;
9:          CAS(&Tail, t, (t.tag+1,x.val))
        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.val).next;
15: *(t.val).next := (x.tag+1,in);
16: Tail := (t.tag+1,in)

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

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

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

Figure 7: Multiprocessor (lock-free) shared queue.

Multiprocessor lock-free queue. In the multiprocessor implementation, an enqueue operation invokes `Enqueue-MW` and passes a list node (*in*) containing the value to be enqueued. The *next* field of the node is initialized in lines 1–2 and the *done* flag cleared in line 3. The *next* field at the end of the queue is then read in lines 4–5. If the comparison at line 6 fails, then another enqueue has completed an operation and a retry occurs. Otherwise, an update of the *next* field read earlier 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 the *Tail* pointer. 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 under consideration must be retried. Lines 8–9 ensure that *Tail* is correctly updated before the retry occurs. Note that it may be the case that the task enqueuing X has not yet updated *Tail*. In this case, if *Tail* were not updated, then the operation in question could be retried repeatedly.

The `Dequeue-MR` procedure returns either a pointer to a list node containing a queue element, or a *nil* pointer, if the queue is empty. To understand how `Dequeue-MR` works, it is important to note that *Head* always points to a dummy node at the head of the list, and if the queue is nonempty, the node *following* the dummy node contains the data to dequeue. 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. In `Dequeue-MR` the *done* flag is initialized at line 17 and then the dummy head node's address is determined at line 18. This address is then compared to that stored in *Tail* at line 19. If equal, then either the list is empty or the node referenced by h has been dequeued and re-enqueued by other concurrent operations. In the latter case, because of the *tag* fields, $h \neq Head$. 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 nonempty. 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. It is easier to attempt a dequeue than to try to determine if an interference has actually occurred.) If $x.val$ 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 head node is read, as discussed above. In line 25, an attempt is made to advance the *Head* pointer past the (old) dummy head 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 moves the data value read at line 24 into the removed node (the old dummy head node). This node is returned at line 28.

Uniprocessor wait-free queue. 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 Fig. 8.

The loop in `Enqueue-MW` (Fig. 7) is unrolled three times to produce the procedure shown in Fig. 8. Three iterations are necessary because some enqueue operation may have chained a new tail node onto the end of the list and gotten preempted before updating *Tail*. In the worst case, the first iteration completes this stalled operation (by updating *Tail* in line 9 in Fig. 7), the second iteration is preempted, and the third iteration is successful. (Notice that only one of these iterations was caused by a preemption. Thus, although there are three loop iterations, there is only one retry, and this scenario is not a violation of (PA).) In Fig. 8, 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 a preemption is detected at any point during these two "iterations," then (PA) ensures that the operation is guaranteed to complete before being preempted again. Therefore, the third iteration can be executed nonpreemptively, 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 Fig. 8. If the operation is preempted, then it can be retried nonpreemptively in the second iteration, which corresponds to lines 35–42.

Comparison. One simple method of comparison is to count the number of shared-memory reads and writes (denoted R and W, respectively) and the number of `CAS` invocations (denoted CAS) in the worst case for one operation (on a uniprocessor). For the multiprocessor version of `Enqueue-MW`, the worst-case path includes *three* loop iterations, with six `CAS` calls being made in lines 7 and 9. In addition, lines 1–2

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

procedure Enqueue-MW(in)
    pm := false;
    x := *(in).next;
    *(in).next := (x.tag+1,nil);
    t := Tail;
    x := *(t.val).next;
    if x.val ≠ nil then
        if CAS(&Tail, t, (t.tag+1,x.val)) then
            t := (t.tag+1,x.val);
            x := *(t.val).next;
            pm := (x.val ≠ nil)
        else
            pm := true
        fi
    fi;
    if ¬pm ∧ t = Tail then
        if CAS(&(*(t.val).next),x,(x.tag+1,in)) then
            CAS(&Tail, t, (t.tag+1,in))
        else
            pm := true
        fi
    fi;
    if pm then
        t := Tail;
        x := *(t.val).next;
        if x.val ≠ nil then
            Tail := (t.tag+1,x.val);
            t := (t.tag+1,x.val);
            x := *(t.val).next
        fi;
        *(t.val).next := (x.tag+1,in);
        Tail := (t.tag+1,in)
    fi
```

```
typedef node:
    record
        val: eltype;
        next: tagged(pointer to node)

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

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

procedure Dequeue-MR() returns pointer to node
    h := Head;
    if h.val = Tail.val then
        if h = Head then
            return nil
        fi
    else
        x := *(h.val).next;
        if x.val ≠ nil then
            out := *(x.val).val;
            if CAS(&Head, h, (h.tag+1,x.val)) then
                *(h.val) := (out,(x.tag+1,nil));
                return h.val
            fi
        fi
    fi
    h := Head;
    if h.val = Tail.val then
        return nil
    else
        x := *(h.val).next;
        out := *(x.val).val;
        Head := (h.tag+1,x.val);
        *(h.val) := (out,(x.tag+1,nil));
        return h.val
    fi
```

Figure 8: Uniprocessor (wait-free) shared queue.

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 to each of the other procedures with the exception of `Enqueue-MW`. It is not clear which path through the code produces the worst-case behavior in this procedure. For this reason, we will consider two possible paths through the code, one of which will be the worst-case path. Fig. 9 summarizes the instruction counts along all considered paths.

Synchronization primitives usually require considerably more cycles than uncached reads and writes. For example, according to LaMarca [13], 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. Although LaMarca's paper dates back to 1994, this and similar architectures are still being used today. Moreover, in embedded systems, older processors are often used for cost reasons.

An admittedly simple method for *approximating* the performance of these algorithms is to take the

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`	preempted	8	4	2	8	4	7	19
1	`Enqueue-MW`	not preempted	5	1	3	5	1	10.5	16.5
1	`Dequeue-MR`		8	3	1	8	3	3.5	14.5

Figure 9: Shared-memory instruction counts along worst-case execution paths for both queue algorithms.

weighted sum of the previous instruction counts. (Since our comparison is only intended to illustrate that some benefit to using a uniprocessor algorithm exists, a precise comparison of the algorithms is unnecessary.) Based on LaMarca's observations, it seems reasonable to assign a weight of 1 to read and write operations, and a weight of 3.5 to `CAS` calls. Figure 9 shows these weighted values. These numbers suggest that the uniprocessor version of `Dequeue-MR` is approximately 84% ($\frac{35}{19} \approx 1.84$) better than its multiprocessor counterpart, and the uniprocessor implementation of `Enqueue-MW` is approximately 31% ($\frac{19}{14.5} \approx 1.31$) better.

For simple lock-free operations that require updating only a single state variable, a uniprocessor implementation may be only a marginal improvement. However, for more complex operations, which update multiple state variables, a uniprocessor implementation should be a considerable improvement. This is illustrated quite well by the improvement of `Dequeue-MR`, which is still a relatively simple operation since it only updates two state variables.

Implementations of operations that involve multiple updates often can be simplified considerably by using the *multi-word-compare-and-swap* (`MWCAS`) primitive, which generalizes `CAS` by allowing multiple words to be accessed. `MWCAS` is impractical to provide in hardware, but fortunately, 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 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, along with an experimental evaluation of its effectiveness.

6.1 The Heuristic

The problem of assigning tasks to supertasks has a strong resemblance to the bin packing problem, which is known to be NP-complete in the strong sense. This is why we focus on using a heuristic. A simplified version of our heuristic is shown in Fig. 10. Tasks are prioritized based upon their object-access patterns and the level of contention associated with each object. We define the *weighted contention* for object ℓ by

Create $|\tau|$ *empty supertasks*;
$R := \tau$;
while $\rho \neq \emptyset \wedge R \neq \emptyset$ **do**
  Select $\ell \in \rho$ *with largest* $R^{[M]}(\ell) \cdot \sum_{T \in \tau} \frac{J_T(\ell)}{T.p}$ *value*;
  $\rho := \rho - \{ \ell \}$;
  while *there exists* $T \in R$ *with* $Q_T(\ell) > 0$ **do**
    Select $T \in R$ *with largest* $Q_T(\ell)$;
    $R := R - \{ T \}$;
    Assign T *to a non-full supertask using the first-fit algorithm*
  od
od;
Add all non-empty supertasks to σ

Figure 10: Heuristic algorithm for assigning tasks to supertasks.

the value $R^{[M]}(\ell) \cdot \sum_{T \in \tau} \frac{J_T(\ell)}{T.p}$. In this expression, $\sum_{T \in \tau} \frac{J_T(\ell)}{T.p}$ approximates the frequency of accesses to ℓ by all tasks in τ. $R^{[M]}(\ell)$ can be thought of as an interference penalty. All tasks that access the object ℓ with the highest weighted contention are assigned to supertasks first. Since task T's interference depends on $Q_T(\ell)$, tasks are assigned to supertasks in nonincreasing order by $Q_T(\ell)$. After all of these tasks are assigned, the remaining objects are considered in nonincreasing order of weighted contention.

Notice that our heuristic assigns tasks to supertasks using the first-fit algorithm. As the name suggests, first-fit assigns each task to the first supertask with sufficient capacity. This strategy actually works against the goal of reducing sharing overhead since it may spread multiple tasks that share the same object across multiple supertasks. However, this strategy ensures that the supertasks are packed *as tightly as possible*, which reduces the inflation overhead when reweighting is used. In short, this aspect of the heuristic targets reweighting inflation rather than object-sharing overhead.

We should point out here that implementing our heuristic is a non-trivial task. Observe that we use task weights to assign tasks to supertasks. However, this assignment affects object-sharing overheads and hence each task's ultimate weight. We address this circularity by applying our heuristic iteratively. In each round, the task weights computed in the previous round (which have been inflated to reflect object-sharing costs) are used when assigning tasks to bins. In addition, the capacity of each bin (supertask) is reduced in each round. Using a bin capacity that is less than one permits a degree of error in the task weight approximations. On the other hand, it may increase the reweighting inflation. In the first round, we bootstrap the iterative process by computing "ideal" weights for the tasks and by setting the bin capacities to one. An ideal weight is based on the assumption that uniprocessor implementations are used for all shared objects. Once tasks are assigned to supertasks, we compute new task weights and check the validity of the assignment. If the cumulative weight of the tasks comprising any supertask exceeds one, then another round is performed. In our experiments, we decremented the bin capacity by 0.01 after each round. We found that the process typically terminated after only two or three rounds.

6.2 Experimental Evaluation

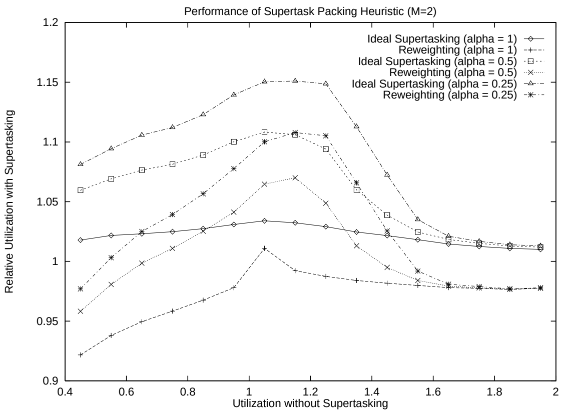
To evaluate our heuristic, we generated 10,000 random task sets for four different system models, consisting of 2, 4, 8, and 16 processors, respectively. Task sets were required to satisfy the assumptions stated earlier in Sec. 4. Our experiments mainly involved lightweight tasks, as we believe such tasks are probably more likely to occur in practice. Our random tasks were generated to have a *base* utilization of at most 0.05. It is important that this utilization not be confused with *actual* utilization, which is only known after the object-sharing overheads have been taken into account. Since object-sharing overheads are often substantial, the actual utilization of a task may be much higher than its base utilization. In addition, the following ranges were used to generate parameters for the random tasks and objects: $|\rho| \in [5, 200]$, $|\tau| \in [25, 25 + \log_2 M]$, $T.p \in [200, 5000]$, $T.e \in [0.05, 0.05 \cdot T.p]$, $Q_T(\ell) \in [0, 5]$, $J_T(\ell) \in [Q_T(\ell), 50]$, $b(\ell) \in [0.000001, 0.1]$, and $r(\ell) \in [0.000001, 0.0142857]$. These ranges were actually chosen rather arbitrarily and hold no particular significance. Our goal here is simply to illustrate that supertasking can be used to reduce lock-free object-sharing overheads. More thorough experimentation would be necessary to express this benefit as a function of the task set parameters.

Results. Our experimental results are shown in Fig. 11. For each sample task set, the weights were computed **(i)** without supertasks, **(ii)** with ideal supertasks, and **(iii)** with reweighted supertasks. To represent the algorithmic improvements of uniprocessor object implementations, we used a scaling factor α, where $R^{[1]}(\ell) = \alpha R^{[M]}(\ell)$ and $B^{[1]}(\ell) = \alpha B^{[M]}(\ell)$ for all $\ell \in \rho$. Although such a scaling factor may be overly simplistic, it nevertheless allows us to demonstrate the benefit of using simpler algorithms. (Observe that $\alpha = 1$ corresponds to the case in which there is no benefit to using an uniprocessor implementation.) After computing all task weights, the schedulability of each sample task set was checked and all task sets that were unschedulable in *any* of the three scenarios were thrown out.

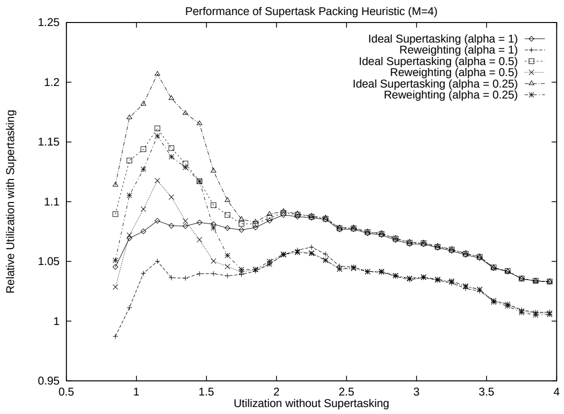
In each graph, the x axis shows the cumulative weight of all tasks when supertasks are not used, while the y axis shows the average relative performance of the heuristic. Here, relative performance is obtained by dividing the cumulative weight without supertasking by the cumulative weight of all supertasks when supertasks are used. Hence, y values larger than one suggest a schedulability improvement.

There are many factors at work that determine the shape of the plotted curves. Unfortunately, the curves for the $M = 8$ and $M = 16$ cases are not smooth. By generating a larger number of sample task sets (which we plan to do) this could presumably be ameliorated. Despite this, we will now briefly explain a few of the dominant factors that influence the shape of each curve.

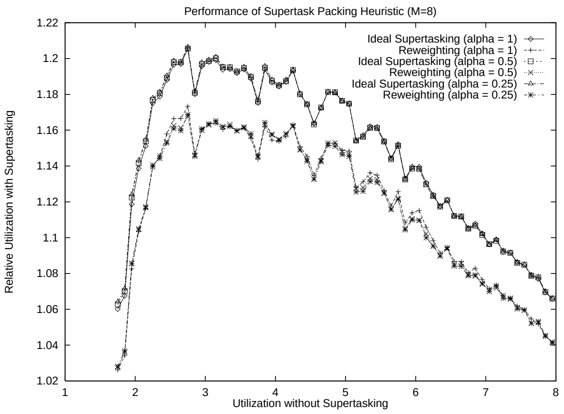
Effectiveness of contention bounds. The performance when using supertasks is only slightly better at very small and very large values of x. This is due to the equations used by Corollaries 1 and 2 to bound worst-case interferences. For very small x, the task sets are small, which results in very little contention. Low contention suggests low object-sharing overhead. As for large x values, the interference term used in



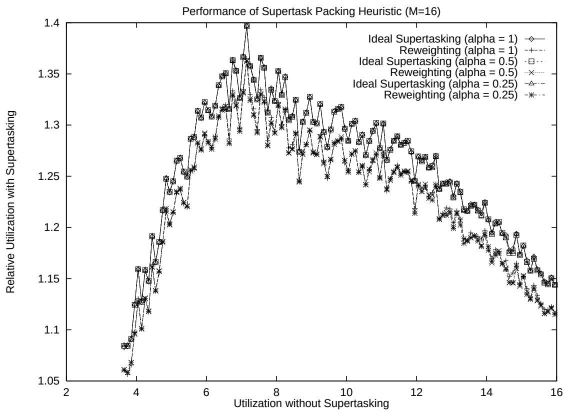
(a) M = 2



(b) M = 4



(c) M = 8



(d) M = 16

Figure 11: Graphs of experimental results for **(a)** 2-, **(b)** 4-, **(c)** 8-, and **(d)** 16-processor systems.

Corollary 1 is a function of the number of *processors* in the system, while that used in Corollary 2 is a function of the number of *supertasks*. In our experimental setup, when x approaches M, the number of supertasks approaches M. Hence, for large values of x, the interference costs computed by Corollaries 1 and 2 are similar.

Effectiveness of heuristic with respect to α. In the $M = 2$ and $M = 4$ graphs, different values of α result in different relative utilizations only around $x = 1$. Around this point, almost all tasks can be assigned to the same supertask, which implies that mostly uniprocessor object implementations will be used. However, as x increases, it becomes less likely that uniprocessor implementations will be used, so the lines representing the different values of α converge to a single line.

These graphs demonstrate that supertasking can be used to reduce object sharing overhead in many cases, particularly as the degree of parallelism in the system increases. Even with reweighting, our heuristic yields an approximate improvement of 15% when $M = 8$ and 30% when $M = 16$. However, as demonstrated by the $M = 2$ graph, the inflation overhead will often dominate any reduction in object-sharing costs when the degree of parallelism is small.

In addition to the *magnitude* of the improvement obtained by applying supertasks, it is also interesting to consider the *frequency* of improvement. For the $M = 2$ case, we found that the number of task sets that showed improvement with the application of reweighted supertasks varied considerably with α. When $\alpha = 1$, 0.5, and 0.25, around 9%, 33.7%, and 40.6% (respectively) of the task sets showed *some* improvement. Furthermore, approximately 82.5% of the task sets considered in the $M = 4$ case showed improvement, while 98.8% showed improvement for the $M = 8$ case, and 99.9% for the $M = 16$ case. These last three percentages suggest that supertasking, even with reweighting, produces an improvement almost always on systems of four or more processors. (However, keep in mind that this observation only applies to the experimental setup considered here.)

7 Conclusion

In this paper, we have addressed the problem of synchronization and object sharing in fair-scheduled multiprocessor systems. To the best of our knowledge, we are the first to consider fair scheduling in multiprocessor systems with non-independent tasks, and the first to consider the use of lock-free objects in real-time multiprocessor systems. We have presented schedulability tests and a technique for assigning Pfair weights to a set of periodic tasks that share lock-free objects. We have also shown that lock-free object-sharing 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.

In future work, we intend to explore some of the lock-based techniques briefly described in Section 3. We also plan to continue developing the supertasking approach, which appears to be the key to reducing synchronization overheads in Pfair-scheduled systems. Specifically, we need to determine whether the safety of component tasks can be guaranteed without reweighting.

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] 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 (OSDI)*, 2000.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.