

# Static-Priority Periodic Scheduling on Multiprocessors\*

Srikanth Ramamurthy<sup>†</sup>  
IBM-Transarc Labs  
11 Stanwix Street  
Pittsburgh, PA 15222

Mark Moir<sup>‡</sup>  
Sun Microsystems Laboratories  
1 Network Drive  
Burlington, MA 01803

## Abstract

*We present a new sufficient condition for the schedulability of preemptable, periodic, hard-real-time task sets using the very simple static-priority weight-monotonic scheduling scheme. Like a previous condition due to Baruah et al., our condition actually determines pfair schedulability. Pfairness requires that the schedule, in addition to being periodic, schedules each task at an approximately even rate. Our condition improves on the previous one in two important ways. First, it can determine that task sets with high utilization and many tasks are schedulable, while the previous condition cannot. Second, our condition applies to both uniprocessors and multiprocessors, while the previous condition applies only to uniprocessors. We present simulations that show that our condition is highly accurate for many cases of interest.*

## 1 Introduction

Schedulability of periodic, preemptable, hard real-time tasks on uniprocessors under static-priority schemes such as rate-monotonic and deadline-monotonic scheduling is well understood; simple and efficient scheduling algorithms are available and widely used [10, 13, 14, 17]. However, the outlook is not as promising in the area of static-priority scheduling algorithms for scheduling periodic tasks on multiple processors. The two main approaches that have emerged for scheduling periodic tasks on multiple processors — partitioning and global scheduling [8] — are described below.

In the *partitioning* approach, each task is fixed to a particular processor. Finding an optimal assignment of tasks to processors is NP-hard [15], so task distribution for partition-

ing schemes is usually done manually, or by non-optimal heuristics. These heuristics typically use a bin-packing algorithm [7, 16] combined with uniprocessor bounds for admission control on each processor [2, 6, 12]. This approach has several inherent limitations. First, because of the use of non-optimal heuristics, an acceptable assignment of tasks to processors may not be found even though one exists. Second, for some schedulable task sets, no such assignment exists, because migration is required in order to schedule them, but this is not possible with the partitioning approach. Finally, the partitioning approach is not suitable for systems that support mode changes [11] (i.e., task sets that change dynamically over time) because of fragmentation. That is, situations can arise in which processor utilization is very low, but a new task cannot be admitted because it will not fit onto a single processor with existing low-utilization tasks.

In the *global scheduling* approach, processors repeatedly execute the highest priority tasks available for execution. It has been shown that using this approach with common priority assignment schemes such as rate monotonic (RM) and earliest deadline first (EDF) can give rise to arbitrarily low processor utilization [8]. Baruah et al. [4, 5] made a significant breakthrough by devising new priority schemes that result in optimal multiprocessor schedulers for hard real-time periodic tasks. However, these are complicated, dynamic-priority scheduling algorithms. As discussed in [3], there are several compelling reasons to consider static-priority scheduling algorithms, despite the fact that the above-mentioned dynamic-priority algorithms can schedule every schedulable task set, while static-priority algorithms cannot. These reasons include simplicity and efficiency, as well as practical concerns for hardware-based implementations.

The focus of this paper is on a static-priority scheme for scheduling periodic, preemptible and migratable hard real-time tasks in multiprocessors. While task migration is often considered to be impractical, in shared-memory symmetric multiprocessors (SMPs), task migration is no more expensive than preempting a task and running it again later on the same processor. One might object that this will result in

\*Author order determined by coin toss.

<sup>†</sup>Email: sri@transarc.com.

<sup>‡</sup>Email: Mark.Moir@sun.com. Work supported in part by an NSF CAREER Award, CCR 9702767, while the second author was at the University of Pittsburgh.

poor cache locality. However, there are no guarantees that a task’s data will still be cached next time it is scheduled, even if it is rescheduled on the same processor, and in fact it is likely that these data will be evicted by other tasks running on the processor in the interim. Thus, at least in the SMP context, guaranteeing predictable behaviour in practice using the partitioning approach seems as difficult as the case in which tasks can migrate. Below we describe the model and problem — which are identical to those considered in [4, 5] — more precisely.

We consider the problem of scheduling sets of preemptible, periodic, hard real-time, migratable tasks in systems of  $m$  identical processors. In the model we consider, tasks are scheduled in discrete quanta and are preempted and/or migrated only at quantum boundaries. Such systems are easier to implement, and allow easier control and analysis of scheduling overhead. Furthermore, in many applications it is not acceptable to preempt tasks at arbitrary points in time [18]. For convenience, we abstract this model into one in which each quantum has a duration of one time unit, and preemptions, migrations, and scheduling decisions can be made instantaneously. We refer to the quantum between time  $i$  and  $i + 1$  as *slot*  $i$ . In this context, we specify a task  $x$  by three integers: its period  $p_x$ , its execution requirement  $e_x$ , and its release time  $r_x$ . Given a set  $T$  of such tasks, our problem is to find a schedule in which each task is scheduled for  $e_x$  slots out of every  $p_x$ , starting at time  $r_x$ . At most one task can be scheduled on each processor during each slot, and each task can be scheduled on at most one processor during each slot.

A particular focus of recent research in this area has been on *proportional share* or *pfair* scheduling [4]. Pfairness requires that the schedule, in addition to being periodic, schedules each task at an approximately even rate. More precisely, we define the *weight*  $w_x$  of task  $x$  to be  $e_x/p_x$ , and require that at time  $r_x + t$ , task  $x$  has been scheduled for either  $\lfloor w_x t \rfloor$  or  $\lceil w_x t \rceil$  slots. Thus,  $x$  is scheduled as closely as possible to its ideal rate, given that tasks are scheduled in integral slots.

Another way to view the pfairness requirement is that each task  $x$  is a periodic task with a period of  $1/w_x$  and an execution requirement of 1. In this case, for each  $j \in \mathbb{N}$ , we could call the interval from  $j/w_x$  to  $(j + 1)/w_x$  a *pseudoperiod* of task  $x$ , and  $(j + 1)/w_x$  a *pseudodeadline*. However, this transformation is not perfect because slots are allocated only at integer values of time, and  $1/w_x$  is not necessarily an integer. Consider for example a task  $x$  with release time  $r_x = 0$ , execution requirement  $e_x = 2$ , and period  $p_x = 5$ . Task  $x$  has weight  $w_x = 2/5$ , and therefore has pseudodeadlines at times 2.5, 5, 7.5, etc. Figure 1 shows two partial schedules for  $x$ . Pseudodeadlines are marked with triangles. In order to be pfair, a schedule must allocate a distinct slot for each pseudoperiod, and the duration of

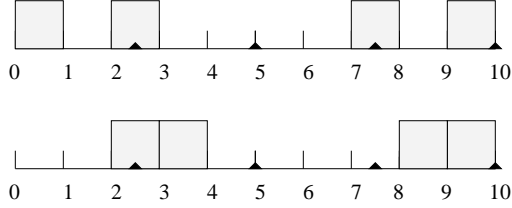
this slot must overlap with the pseudoperiod. Thus, the first schedule shown is pfair, while the second is not, because no slot that overlaps with  $x$ ’s third pseudoperiod is allocated to  $x$ . (Observe that at time  $t = 8$ ,  $x$  has been scheduled only twice, but  $\lfloor w_x t \rfloor = 3$ .) Because pseudodeadlines are not necessarily integers, a particular slot can be allocated to either of two adjacent pseudoperiods. For example, slot 2 is allocated to  $x$ ’s first pseudoperiod in the top schedule, and to its second pseudoperiod in the bottom one. (Although the bottom schedule later becomes not pfair, it is still pfair at time 3.) As will be seen later, this fact complicates the proof of our result, which is discussed next.

In this paper we present a new sufficient condition for the pfair schedulability of task sets using the weight monotonic (WM) scheduling scheme. WM [3] is a very simple, static-priority scheme that gives higher priority to tasks of higher weight. To our knowledge, the only previous such condition is due to Baruah [3]. Baruah’s condition states that for a set  $T$  of  $n$  tasks, if  $\sum_{x \in T} w_x \leq \sum_{j=n}^{2n-1} \frac{1}{j}$ , then  $T$  is scheduled in a pfair manner by WM on one processor. This function quickly approaches  $\ln 2$  as  $n$  increases. Therefore, this condition cannot predict schedulability of large task sets that utilize the processor more than about 70%. In contrast, our condition examines individual task weights, rather than basing its decision solely on the total weight of all tasks. Simulation experiments described in Section 4 show that our condition can accurately predict the schedulability of task sets with thousands of tasks and total utilization in the 80% to 90% range. (Our experiments show that very few task sets with utilization in the 90% to 100% utilization range are schedulable by WM.)

We show that for a set of tasks  $T$ , with tasks labeled in order of decreasing weights, if for every  $x$  in  $T$  there exists a  $t \in (0, \lfloor \frac{1}{w_x} \rfloor]$  such that  $\sum_{y < x} \lceil w_y t \rceil < mt$ , then  $T$  is scheduled in a pfair manner by WM on  $m$  processors. This condition can be evaluated for a set of  $n$  tasks in  $\Theta(n \lfloor \frac{1}{w_x} \rfloor)$  time, where  $x$  is the task of lowest weight.

Below we discuss motivation and related work. There are several advantages to considering pfair or proportional share scheduling rather than the usual periodic scheduling requirement. First, real-time tasks are given stronger timeliness guarantees. This may be useful for example in reducing jitter in a videoconferencing application. Second, as pointed out by Stoica *et al.* [18], real-time and non-real-time tasks can be treated simply and equally by giving each task a “share” of the processor. (Stoica *et al.* considered only uniprocessor systems, but the same benefits clearly apply to multiprocessor systems.)

However, this paper demonstrates a more fundamental reason to consider pfair scheduling as opposed to standard periodic scheduling: by maintaining a bound on the difference between how much time has been allotted to each task



**Figure 1.** Two partial schedules for a single task  $x$  with  $e_x = 2$ ,  $p_x = 5$ , and  $r_x = 0$ . Pseudodeadlines are indicated by black triangles. Grey boxes indicate slots in which  $x$  is scheduled. The first schedule is pfair; the second is not.

and the amount of time that would have been allotted to the task if it progressed uniformly at its ideal rate, we make it easier to maintain this property in the future. A good analogy is in inductive proofs: we often prove a property that is stronger than the one in which we are really interested in order to facilitate the inductive step. It was this insight that allowed us to achieve our scheduling condition. Indeed, as explained later, we consider a property that is even stronger than pfairness in order to obtain our condition. The stronger property allows us to overcome the key problem faced in deriving our condition: when tasks can migrate between processors, uniprocessor conditions for periodic scheduling such as the one presented in [14] cannot be extended to multiprocessors because the counting arguments cannot account for the fact that a task may not be allotted multiple slots at any given instant. The strengthened pfairness property directly implies that a task cannot be scheduled more than once during a single slot, allowing us to obtain a scheduling condition for multiprocessors.

As stated earlier, Baruah *et al.* [4] proved that any task set whose combined weights is at most  $m$  can be scheduled in a pfair manner on  $m$  processors, and presented a scheduling algorithm that would achieve this. Later, Baruah *et al.* presented a more efficient algorithm [5]. Both of these are dynamic-priority scheduling algorithms; the algorithm (WM) considered in this paper is a static-priority algorithm, and is therefore considerably simpler, more efficient, and easier to implement.

Before presenting our condition, a clarification of the precise definition of static- and dynamic-priority scheduling algorithms is in order. Baruah [3] defines a scheduling algorithm to be static iff for any pair of tasks  $x$  and  $y$ , whenever both tasks are “contending”, the algorithm gives priority to the same task. Baruah defines contending informally as follows: “a task is *contending* at a given time instant if it can be allocated the processor without violating any constraints”. We do not believe it makes sense to have the distinction between static- and dynamic-priority algorithms depend on the problem they are used to solve. For example, by the above definition, WM is a static-priority algorithm for pfair scheduling but not for periodic schedul-

ing, despite the fact that it is the same algorithm producing the same schedule for any given task set (recall that pfair implies periodic).

We therefore propose the following characterization of static scheduling algorithms. A *static-priority scheduling algorithm* is one in which there is some function  $contending(x, t)$  that can be evaluated using only the scheduling history of task  $x$  at time  $t$  and local information of task  $x$ <sup>1</sup>, such that at every time  $t$  at which  $contending(x, t)$  and  $contending(y, t)$  both hold, priority is given to the same task. We believe this captures the spirit of static-priority scheduling, and is not problem-dependent. In particular, WM is a static-priority scheduling algorithm for both pfair and periodic scheduling using this definition.

Whether or not we consider WM to be a static-priority algorithm (and we argue that we should), it is clear that it is very simple, efficient, and easy to implement, and is therefore of considerable interest in practice.

The remainder of the paper is organized as follows. In Section 2 we formally define the pfair multiprocessor scheduling problem, and the WM scheduling algorithm. Then, in Section 3 we prove that our condition is sufficient for a task set to be scheduled in a pfair manner by WM. In Section 4, we present our simulation experiments and their results. Concluding remarks appear in Section 5.

## 2 Preliminaries

In this section, we define tasks and task sets, and schedules and pfair schedules, and present the simple static-priority scheduling scheme (WM) for which we prove a scheduling condition in the following section.

**Definition 1:** A task  $x$  is characterized by three integers: its period  $p_x$ , its per-period worst-case execution requirement  $e_x$ ,  $0 < e_x \leq p_x$ , and its release time  $r_x \geq 0$ . The weight of task  $x$  is  $w_x = e_x/p_x$ . Note that  $0 < w_x \leq 1$ . We

<sup>1</sup>Local information of task  $x$  could include environmental information such as release times, period, execution requirement, etc. Thus, this definition also applies to other scheduling requirements, such as aperiodic, sporadic, etc.

consider a set  $T$  of such tasks with distinct labels such that  $y < x \Rightarrow w_y \geq w_x$ .

**Definition 2:** An  $m$ -processor schedule for a task set  $T$  is a function  $S : T \times \mathbb{N} \rightarrow \{0, 1\}$ , such that  $(\forall t \in \mathbb{N} :: \sum_{x \in T} S(x, t) \leq m)$  and  $(\forall x \in T \forall t \in \mathbb{N} :: t < r_x \Rightarrow S(x, t) = 0)$ . A *schedule prefix of size  $t$*  is a function  $S : T \times [0..t) \rightarrow \{0, 1\}$ , such that  $(\forall t \in [0..t) :: \sum_{x \in T} S(x, t) \leq m)$  and  $(\forall x \in T \forall t \in [0..t) :: t < r_x \Rightarrow S(x, t) = 0)$ . We say a schedule prefix  $S'$  of length  $u$  is a *prefix of schedule  $S$*  iff  $(\forall x \in T \forall t \in [0, u) :: S'(x, t) = S(x, t))$ .

For the remainder of the paper, we assume a given task set  $T$  and  $m$  processors, and simply use “schedule” to denote an  $m$ -processor schedule for  $T$ .

**Definition 3:** If  $S$  is a schedule or schedule prefix of length at least  $u$ , for  $x \in T$ ,  $t \in [0, u)$ , then

$$C_S(x, t) = \begin{cases} 0 & \text{if } t < r_x \\ \sum_{u=r_x}^{t-1} S(x, u) & \text{otherwise} \end{cases}$$

Below we give a formal definition of fairness. The intuition for this definition is as follows. Recall from Section 1 that we view a task  $x$  as having pseudoperiods of length  $1/w_x$ , each of which must be satisfied by allocating a slot whose duration overlaps the pseudoperiod. This can be restated as follows: for any time  $t$ , the end of the next pseudoperiod to be satisfied is after  $t$  (i.e.,  $x$  has not missed its next pseudodeadline), and the beginning of the most recently satisfied pseudoperiod is before  $t$  (i.e.,  $x$  has not already satisfied a pseudoperiod that has not yet started). Recall that each pseudoperiod of task  $x$  is  $1/w_x$  time units long. Thus, the  $i$ th pseudoperiod covers the interval from  $i/w_x$  to  $(i + 1)/w_x$ . Therefore, given the above definition of  $C_S(x, t)$ , at time  $t$  in a schedule  $S$ , the next pseudoperiod to be satisfied by task  $x$  covers the interval from  $C_S(x, t)/w_x + r_x$  to  $(C_S(x, t) + 1)/w_x + r_x$ , and the most recent pseudoperiod already satisfied (if it exists) covers the interval from  $(C_S(x, t) - 1)/w_x + r_x$  to  $C_S(x, t)/w_x + r_x$ . Therefore, the intuition described above can be formalized as follows.

**Definition 4:** A schedule  $S$  is *pfair* iff  $(\forall x \in T \forall t \in \mathbb{N} : t \geq r_x :: C_S(x, t) < w_x(t - r_x) + 1 \wedge C_S(x, t) > w_x(t - r_x) - 1)$ .

We next define what it means for a task  $x$  to be *valid*<sup>2</sup> at time  $t$  in a schedule  $S$ . The intuition is that if  $\text{valid}_S(x, t)$  does not hold, then  $x$  has already been scheduled enough times to satisfy all pseudodeadlines before time  $t + 1$ . Thus, if  $x$  is scheduled at time  $t$ , the pseudoperiod for which  $x$  is

<sup>2</sup> $\text{valid}(x, t)$  is the “contending” function by which we would characterize WM as a static-priority scheduling algorithm (see Section 1).

scheduled at time  $t$  will not overlap the slot between times  $t$  and  $t + 1$ , and therefore  $x$  will violate pfairness.

**Definition 5:** Task  $x$  is *valid at time  $t$  in schedule prefix  $S$*  of length at least  $t$ , denoted  $\text{valid}_S(x, t)$  iff  $C_S(x, t) < w_x(t + 1 - r_x)$ . (Note, by definition, for  $x \in T$  and  $t < r_x$ ,  $\neg \text{valid}(x, t)$  holds.)

Below we informally describe the WM scheduling algorithm, and then formally specify the schedule constructed by this algorithm. Our specification of WM is a natural extension of the uniprocessor version [3]. It is cosmetically different in that our definition of  $\text{valid}(x, t)$  appears different to the definition of *contending* in [3]. However, these two definitions are in fact equivalent. The difference is a result of the need to eliminate the floor/ceiling arithmetic that has plagued other recent papers in this area in order to acquire a simple proof of a scheduling condition that is tight enough to be useful.

**Notation:** Following [9], we use  $[P]$ , where  $P$  is a predicate, to denote 0 if  $P$  is false and 1 if it is true.  $\square$

**Algorithm WM** WM schedules at each time  $t \in \mathbb{N}$  the  $m$  tasks of highest weight amongst tasks that are valid at time  $t$  (or all valid tasks if there are fewer than  $m$ ).  $\square$

Below we specify a schedule for a given task set. It is easy to see that the specified schedule is the one constructed by WM for the task set.

**Constructing a pfair schedule:** We specify a schedule inductively. Schedule prefixes of length 0 are vacuous. Given a schedule prefix  $S^u$  of length  $u$ , we specify schedule prefix  $S^{u+1}$  as follows. For  $x \in T$  and  $t \in [0, u)$ ,  $S^{u+1}(x, t) = S^u(x, t)$  and  $S^{u+1}(x, u) = [\text{valid}_{S^u}(x, u) \wedge \sum_{y \in T} [\text{valid}_{S^u}(y, u) \wedge y \leq x] \leq m]$ . We call the resulting schedule  $S_A$ .  $\square$

### 3 Scheduling Condition

In this section, we present and prove our scheduling condition. Before proceeding with our proof, a high-level, intuitive explanation is in order. Our strategy is to slightly strengthen the pfairness property, and to then prove that if  $S_A$  does not satisfy the strengthened property, then our scheduling condition does not hold. The strengthened property is shown below.

**Definition 6:** A schedule  $S$  is *strong-pfair* iff  $(\forall x \in T \forall t \in \mathbb{N} : t \geq r_x :: C_S(x, t) < w_x(t - r_x) + 1 \wedge C_S(x, t) > w_x(t + 1 - r_x) - 1)$ .

Observe that the second conjunct has been strengthened to require that, at any time  $t$ ,  $x$ 's next pseudodeadline to be satisfied falls after time  $t + 1$ , instead of after time  $t$  as was the case in the original definition. The reason for the strengthening is explained later. To achieve our condition, we consider the first time  $t$  at which some task violates this property, and some task  $x$  that does so at time  $t$ . We then define the *busy point* of  $x$  with respect to  $t$ , denoted  $B(x, t)$ , as the earliest time before  $t$  such that in every slot between  $B(x, t)$  and  $t$ , every processor is allocated to some task of higher priority than  $x$ . Next we consider any  $u \in (B(x, t), t]$ , and derive a bound on the number of times each such task can be scheduled between the busy point and  $u$ , and this yields our main scheduling condition. We also prove that, if  $x$  violates the condition at time  $t$ , then  $x$  has been valid and not scheduled for at least the last  $\lfloor \frac{1}{w_x} \rfloor$  slots. This implies that in all of these slots, all processors were allocated to higher priority tasks than  $x$ , which implies that the interval between  $B(x, t)$  and  $t$  is at least  $\lfloor \frac{1}{w_x} \rfloor$  slots wide. Thus, our condition is proved for at least  $\lfloor \frac{1}{w_x} \rfloor$  values of  $u$ . As a result, any task set that does not satisfy our condition for *all* such values is schedulable.

We now explain the intuition behind the bound on the number of times a higher priority task  $y$  can be scheduled between  $B(x, t)$  and  $u$ , for some  $u \in (B(x, t), t]$ . Because the definition of *valid* ensures that a task  $y$  never satisfies a pseudoperiod before that pseudoperiod begins, the maximum number of times  $y$  can be scheduled between times  $B(x, t)$  and  $u$  is the number of pseudoperiods that overlap this interval, which is  $\lceil w_y(u - B(x, t)) \rceil + 1$ . Unfortunately, however, this counting method does not yield a useful condition: we need a smaller bound. We achieve this by observing that if  $\lceil w_y(u - B(x, t)) \rceil + 1$  pseudoperiods do overlap the interval between  $B(x, t)$  and  $u$ , then the first and last ones must overlap by less than one time slot. Therefore, this case would arise only if a task is forced to satisfy a pseudoperiod at time  $B(x, t)$  whose pseudodeadline falls between  $B(x, t)$  and  $B(x, t) + 1$ . The intuition is that, in such a case,  $x$  is very close to missing a pseudodeadline, even if it does not in fact miss it. We strengthen the pfairness property exactly enough to preclude this possibility. For tasks with relatively low weight (and therefore long pseudoperiods), this restriction is of little consequence, because the task has relatively many slots before the last one in each pseudoperiod in which to satisfy that pseudoperiod. Also, in a task set where some task is forced to satisfy some pseudoperiod in its last possible slot, it is likely that the same task will miss a later pseudodeadline anyway, and thus our condition does not miss a schedulable task set in this case. For tasks of greater weight (and therefore shorter pseudoperiods), however, the restriction is more severe. The simulation studies presented in the next

section clearly show this effect.

After presenting the proof of our main condition, we further strengthen the condition by considering the special case of two tasks with total weight at most 1. We show that all such tasks sets are scheduled in pfair manner by WM.

We now proceed with our proof. Due to lack of space, we defer proofs of some lemmas to the full paper. Our first lemma says that at most  $m$  tasks are scheduled at a time.

**Lemma 1:** For all  $t \in \mathbb{N}$ ,  $\sum_{z \in T} S_A(z, t) \leq m$ .

The next lemma says that no task gets too far ‘‘ahead’’ of its ideal rate. The intuition is that if scheduling task  $x$  at time  $t' < t$  would satisfy a pseudoperiod that does not begin before  $t' + 1$  (and therefore before  $t$ ), then  $x$  is invalid at time  $t$ , and is therefore not scheduled.

**Lemma 2:** For  $x \in T$ ,  $t \in \mathbb{N}$ , if  $t \geq r_x$ , then  $C_{S_A}(x, t) < w_x(t - r_x) + 1$ .

Our next lemma is used to bound the number of slots allocated to a task  $y$  in a given interval, assuming that the strengthened pfairness property has not yet been violated. We first define  $D(y, B, u)$  as the number of slots allocated to task  $y$  between times  $B$  and  $u$ , and then prove that, if  $y$  has been scheduled at least  $w_y(B - r_y)$  times before  $B$ , then  $D(y, B, u)$  cannot exceed  $\lceil w_y(u - B) \rceil$ . The intuition is that if  $y$  has been scheduled at least  $w_y(B - r_y)$  times before  $B$ , then  $y$  has already satisfied all pseudoperiods that finish before  $B$ . Thus, there are at most  $\lceil w_y(u - B) \rceil$  remaining pseudoperiods that overlap slots between  $B$  and  $u$ , and as tasks are not scheduled while they are invalid, they do not satisfy pseudoperiods that begin after  $u$  in this interval.

**Definition 7:** For  $y \in T$ ,  $B \in \mathbb{N}$ , and  $u \in \mathbb{N}$ ,  $u \geq B$ , define  $D(y, B, u) = C_{S_A}(y, u) - C_{S_A}(y, B)$ .

**Lemma 3:** For  $y \in T$  and  $B \in \mathbb{N}$ , and  $v \in \mathbb{N}$ ,  $v \geq B$ , if  $C_{S_A}(y, B) \geq w_y(B - r_y)$ , then  $D(y, B, v) \leq \lceil w_y(v - B) \rceil$ .

We next define  $B(x, t)$ , and then prove a lemma that says that the sum over all tasks with higher priority than  $x$  of the number of slots allocated to these tasks in the interval from  $B(x, t)$  until time  $t$  is  $m$  times the width of this interval. The intuition is straightforward, given the definition of  $B(x, t)$ , and the fact that the scheduling algorithm schedules at most  $m$  tasks in each time slot.

**Definition 8:** Define  $B(x, t) = \max\{t' \in \mathbb{N} \mid t' < t \wedge \sum_{y < x} S_A(y, t') < m\} + 1$ . (Let us consider  $S_A(y, -1)$  to be 0 so that  $B(x, t)$  is well-defined and non-negative.)

**Lemma 4:** For  $x \in T$ ,  $t \in \mathbb{N}$ , and  $v \in (B(x, t), t]$ ,

$$\sum_{y < x} D(y, B(x, t), v) = m(v - B(x, t)).$$

**Proof:** By Definitions 3 and 7,

$$D(y, B(x, t), v) = \sum_{t'=B(x,t)}^{v-1} S_A(y, t').$$

Thus,

$$\begin{aligned} \sum_{y < x} D(y, B(x, t), v) &= \sum_{y < x} \sum_{t'=B(x,t)}^{v-1} S_A(y, t') \\ &= \sum_{t'=B(x,t)}^{v-1} \sum_{y < x} S_A(y, t'). \end{aligned}$$

By Definition 8 and Lemma 1, for  $t' \in [B(x, t), v)$ ,

$$\sum_{y < x} S_A(y, t') = m.$$

Thus,

$$\sum_{y < x} D(y, B(x, t), v) = m(v - B(x, t)). \quad \square$$

The next two lemmas show that  $x$  is valid but not scheduled for at least  $\lfloor \frac{1}{w_x} \rfloor$  slots before  $t$ . As explained above, this shows that the interval between  $B(x, t)$  and  $t$  is at least  $\lfloor \frac{1}{w_x} \rfloor$  slots wide. The intuition for the first lemma is essentially that if  $x$  is scheduled once per pseudoperiod, then  $x$  remains pfair.

**Lemma 5:** For a schedule  $S$ ,  $x \in T$  and  $t \in \mathbb{N}$ , if  $C_S(x, t - \lfloor \frac{1}{w_x} \rfloor) \geq w_x(t - \lfloor \frac{1}{w_x} \rfloor) + 1 - r_x - 1$  and  $C_S(x, t) < w_x(t + 1 - r_x) - 1$  and  $t' \in [t - \lfloor \frac{1}{w_x} \rfloor, t)$ , then  $S(x, t') = 0$ .

The intuition for the next lemma is that  $x$  is valid at time  $t'$  if it has not yet satisfied the pseudoperiod(s) that include time  $t$ .

**Lemma 6:** For a schedule  $S$  and  $x \in T$ , if  $C_S(x, t) < w_x(t + 1 - r_x) - 1$ , and  $t' \in [t - \lfloor \frac{1}{w_x} \rfloor, t)$ , then  $valid_S(x, t')$  holds.

The next lemma formalizes the counting technique described above, and makes an algebraic substitution that allows our condition to be independent of  $B(x, t)$ , which is of course not available to the condition checker.

**Lemma 7:** For  $x \in T$  and  $t \in \mathbb{N}$ , if for all  $y < x$ ,  $C_{S_A}(y, B(x, t)) \geq w_y(B(x, t) - r_y)$  holds, then

$$(\forall u \in (0, t - B(x, t)) :: \sum_{y < x} [w_y u] \geq mu).$$

**Proof:** Lemma 3 implies that, for any  $v \geq B(x, t)$ ,

$$\sum_{y < x} D(y, B(x, t), v) \leq \sum_{y < x} [w_y(v - B(x, t))].$$

By Lemma 4 and Definition 7, this implies that for all  $v \in (B(x, t), t]$ ,

$$\sum_{y < x} [w_y(v - B(x, t))] \geq m(v - B(x, t)).$$

The lemma follows by substituting  $u = v - B(x, t)$ .  $\square$

The last lemma is the main lemma used to prove our scheduling condition. It says that if some task violates the stronger pfairness property, then our scheduling condition does not hold. The key part of this proof is to observe that, by the definition of  $B(x, t)$ , not all processors are allocated to tasks with higher priority than  $x$  at time  $B(x, t) - 1$ . Thus, each such task  $y$  is either invalid or is scheduled at that time. We show that in either case, at time  $B(x, t)$ ,  $y$  has satisfied all pseudoperiods that begin before  $B(x, t)$ . As explained above, this allows us to achieve a tighter bound on the number of times  $x$  is scheduled between  $B(x, t)$  and some  $u \in (B(x, t), t]$ .

**Lemma 8:** If  $(\exists x \in T \exists t \geq r_x :: C_{S_A}(x, t) < w_x(t + 1 - r_x) - 1 \vee C_{S_A}(x, t) \geq w_x(t - r_x) + 1)$ , then

$$(\exists x \in T \forall u \in (0, \lfloor \frac{1}{w_x} \rfloor] :: \sum_{y < x} [w_y u] \geq mu).$$

**Proof:** By Lemma 2, for some  $x \in T$  and  $t \geq r_x$ ,  $C_{S_A}(x, t) < w_x(t + 1 - r_x) - 1$ . Choose  $x$  and  $t$  such that

$$\begin{aligned} t \geq r_x \wedge C_{S_A}(x, t) &< w_x(t + 1 - r_x) - 1 \wedge \\ (\forall y \in T \forall t' < t :: \\ t' < r_y \vee C_{S_A}(y, t') &\geq w_y(t' + 1 - r_y) - 1). \end{aligned} \quad (1)$$

By Lemmas 5 and 6, we have

$$(\forall t' \in [t - \lfloor \frac{1}{w_x} \rfloor, t) :: S_A(x, t') = 0 \wedge valid_{S_A}(x, t')). \quad (2)$$

By the construction of  $S_A$  and Definition 8, this implies that  $B(x, t) \leq t - \lfloor \frac{1}{w_x} \rfloor$ . Also, because  $B(x, t) - 1 < t$ , (1) implies that  $C_{S_A}(y, B(x, t) - 1) \geq w_y(B(x, t) - r_y) - 1$  holds for all  $y \in T$ . This in turn implies that for all  $y$ , if  $S_A(y, B(x, t) - 1) = 1$ , then  $C_{S_A}(y, B(x, t)) \geq w_y(B(x, t) - r_y)$  holds. Also, for all  $y < x$ , if  $S_A(y, B(x, t) - 1) = 0$ , then by Definition 8 and the construction of  $S_A$ ,  $\neg valid_{S_A}(y, B(x, t) - 1)$  holds. Thus,  $C_{S_A}(y, B(x, t)) = C_{S_A}(y, B(x, t) - 1) \geq w_y(B(x, t) - r_y)$  holds in this case too. Thus, by Lemma 7, the lemma holds. (Observe that  $t - B(x, t) \geq \lfloor \frac{1}{w_x} \rfloor$ .)  $\square$

We now state the condition for the special 2-task case. Due to lack of space, we defer the proof to the full paper. The proof is similar in structure to the preceding proof, but simpler.

**Lemma 9:** If  $T = \{x, y\}$  such that  $w_x + w_y \leq 1$ , then  $S_A$  is pfair.

Lemmas 8 and 9, together with Definition 4, yield the following.

**Theorem 1:** If

$$(|T| = 2 \wedge \sum_{x \in T} w_x \leq 1) \vee$$

$$(\forall x \in T \exists t \in (0, \lfloor \frac{1}{w_x} \rfloor] :: \sum_{y < x} \lceil w_y t \rceil < mt),$$

then  $S_A$  is pfair.

## 4 Evaluation

In this section we describe simulation experiments we conducted in order to evaluate the accuracy of our condition, and discuss the results of these experiments.

We conducted experiments for 1, 2, 4, 8, 16, and 32 processors. For 1 processor, we compare the number of task sets that are correctly scheduled by WM and the number of task sets identified as schedulable by our condition and by Baruah’s condition [3]. For multiple processors, we evaluate only our condition, as Baruah’s condition does not apply in this case. Below we describe our experimental methodology, and then discuss the results.

### 4.1 Experimental Methodology

We first describe the overall structure of our experiments. For each experiment, we generated 10000 task sets and categorized each one by total utilization into one of 100 “buckets”. The first bucket contains task sets that have utilization less than 1%, the second bucket contains task sets that have utilization at least 1% but less than 2%, and so on. For each bucket, we recorded the total number of task sets generated, the total number of task sets scheduled by WM, the total number of task sets predicted as schedulable by our condition, and, in the case of 1 processor, the total number of task sets predicted as schedulable by Baruah’s condition.

In the graphs shown in Figures 2 through 4, the  $x$  axis shows the buckets described above. The  $y$  axis shows a percentage of task sets. For example, a point on the “scheduled” curve at (0.2,100) shows that 100% of all task sets generated in this experiment that had total utilization at least 19% and less than 20% were scheduled by WM.

It remains to describe how we generated task sets. Each experiment (one per graph shown) was specified by three parameters:  $A$ ,  $F$ , and  $m$ .  $m$  is the number of processors, and  $A$  and  $F$  determined how individual task weights were generated, as described below. For each task set, we chose a total utilization target from a uniform distribution between

0 and 1, and then attempted to create a task set with this utilization. This was achieved by repeatedly adding a new task until the total utilization exceeded the target. Thus, our task sets did not have exactly the target utilization, but this mechanism had the desired effect of spreading the total utilization of the generated task sets over the entire range where possible. We discarded any task set that did not have more than  $m$  tasks, as such task sets are trivially schedulable, and trivially predicted by both our condition and by Baruah’s in the 1-processor case. It remains to describe the mechanism we used to generate individual task weights.

We wanted to be able to test a wide range of task sets, with varying task weight distributions. In particular, we wanted to avoid the possibility that all task sets would be made up of similar weight tasks. Therefore, we combined two task weight distributions. With probability  $F$  we simply chose a task weight from a uniform, random distribution from 0 to 1. With probability  $1-F$ , we first conducted 29 binomial experiments<sup>3</sup> with probability of success equal to  $A$ . We divided the total number of successes by 29 to give us a weight between 0 and 1. We then added a random number from a uniform distribution between  $-1/29$  and  $1/29$  in order to avoid a quantization effect on task weights. We discarded tasks whose weights were not greater than zero and less than one.

We first conducted experiments for 1, 2, 4, 8, 16, and 32 processors with  $F = 0.1$ . Thus, in these experiments, it is expected that only 10% of task weights are chosen from a uniform distribution between 0 and 1. For each number of processors and value of  $F$ , we conducted experiments for  $A = 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8$ , and  $0.9$ . Thus we tested task sets made up primarily of tasks with very low weights ( $A = 0.01$ ) and weights ranging from relatively low ( $A = 0.1$ ) to relatively high ( $A = 0.9$ ). In fact, with  $A = 0.9$ , the experiments did not complete because of the low probability of finding a task set with more than  $m$  tasks of very high weight. We could have designed a special-case task set generator to collect data for this case, but given the results we did collect, this did not seem worthwhile.

As discussed below, the results for  $F = 0.1$  were very encouraging. In order to explore the sensitivity of our results to  $F$ , we also conducted experiments for 32 processors using  $F = 0.2$  and  $F = 0.4$ . Finally, we conducted experiments for 1, 2, 4, 8, 16, and 32 processors with  $F = 1$ . (In this case, all task weights are chosen from a uniform distribution.) Due to limited space, we are unable to present the results of all of these experiments here. Instead, we present some interesting cases, and discuss general trends observed in the other experiments.

<sup>3</sup>The reason for this choice seems to be lost to history.

## 4.2 Results

Figures 2 and 3 show the results of the experiments with 32 and 1 processors, respectively, with  $F = 0.1$ . Each figure shows one graph for each value of  $A$  tested.

First, let us examine the results for our 32-processor experiments, shown in Figure 2. In the  $A = 0.01$  case, our condition is virtually indistinguishable from the optimal condition (i.e., the one that perfectly predicts schedulability<sup>4</sup>). Observe that even in the 80% to 90% utilization range, almost all task sets are schedulable in all cases and are correctly predicted as such by our condition. In the 90% to 100% utilization range, almost no task sets are schedulable, but those that are schedulable are correctly predicted by our condition.

As we increase  $A$ , we start to see a more noticeable difference between our condition and the ideal one. Nonetheless, our condition continues to track the optimal one very well up to relatively high utilization ranges and values of  $A$ . As discussed in Section 3, we expected our condition to fare worse with higher task weights, and our experiments show that this is indeed the case. In particular, observe that in the  $A = 0.5$  case for 32 processors, most task sets in the 75%-80% utilization range are schedulable, while our condition predicts almost none of them. (This is the case in which our condition performed worst among all of our experiments.) For larger values of  $A$ , WM is able to schedule fewer task sets in this range, so the performance of our condition, relative to the ideal one, improves. In almost all cases, either very few task sets are schedulable, or our condition correctly predicts a substantial fraction of those that are. Observe that when  $A$  reaches 0.8, no task sets are schedulable. This is not surprising, as the task sets in such cases are made of many high-weight tasks; such task sets usually require a dynamic scheduling scheme because we have many similar tasks, each of which can be scheduled during many consecutive slots. Static-priority scheduling schemes treat similar-weight tasks very differently in such cases, and are therefore unlikely to succeed.

The results for 2, 4, 8, and 16 processors (not shown) showed similar trends. For fewer processors, both WM and our condition performed slightly better.

Let us now examine the uniprocessor results. In Figure 3, we again compare our condition to the optimal one; we also include Baruah’s uniprocessor condition. As in the multiprocessor experiments, our condition tracks the optimal one extremely well for values of  $A$  up to about 0.3, and then begins to do slightly worse. However, it is interesting to note that in this case, when  $A \geq 0.5$ , almost all task sets are schedulable and our condition correctly predicts this. This is due to the special case in our condition. Because

<sup>4</sup>For convenience, we use “schedulable” to mean “scheduled in a pfair manner by WM”.

we have only one processor in this case, when we generate tasks from a distribution with average weight greater than 0.5, virtually all acceptable task sets (recall that we discard those that have total weight greater than  $m$  or fewer than  $m + 1$  tasks) have exactly two tasks. Before we added the special case to the condition, our condition deteriorated quickly in these cases. Similar reasoning suggests that almost all acceptable task sets in the  $A = 0.7, m = 2$  case would have exactly 3 tasks. It is therefore tempting to conjecture that our special case can be generalized to say that all sets of  $m + 1$  tasks with total weight at most  $m$  can be scheduled by WM on  $m$  processors. However, this is not the case. For example, a set of three tasks with weights  $7/10$ ,  $7/10$ , and  $6/10$  is not schedulable by WM on 2 processors. (In fact most such task sets were not schedulable by WM.)

Observe that in all cases, our condition correctly predicts significantly more of the schedulable task sets than Baruah’s condition does. The results obtained from Baruah’s condition are consistent with what we would expect, given that the condition is a function only of the total utilization and the number of tasks, rather than of individual task weights. For example, when  $A = 0.01$ , Baruah’s condition predicts very few task sets in the 70% to 80% range, and none in the 80% to 100% range. This is not surprising, as the utilization bound of Baruah’s condition is less than 0.8 for three or more tasks. Clearly the task sets generated in this experiment with utilization in the 80% to 100% range are very unlikely to have fewer than three tasks.

Figure 4 shows the results for 1, 2, and 32 processors using  $F = 1$ . (That is, in these experiments, all task weights are chosen from a uniform distribution between 0 and 1.) Given that our condition is at a disadvantage when faced with tasks of relatively high weight, it (and the algorithm itself) perform surprisingly well in this case. The results for 4, 8, and 16 processors were similar.

## 5 Concluding Remarks

We have presented a trivial extension to the static-priority weight monotonic scheduling algorithm of Baruah [3] to allow it to schedule sets of hard-real-time, preemptible, periodic tasks on multiprocessors. We have also provided a sufficient condition for the schedulability of task sets using this algorithm. Our simulations indicate that, for task sets consisting of tasks that require a relatively low fraction of a processor, the algorithm is very effective and that our condition is highly accurate in predicting this. These experiments also show that our condition substantially improves on the accuracy of Baruah’s sufficient condition for uniprocessors. The algorithm and our condition both deteriorate as the average utilization of individual tasks increases. Nonetheless, in many applications, most tasks require a relatively small fraction of a processor.

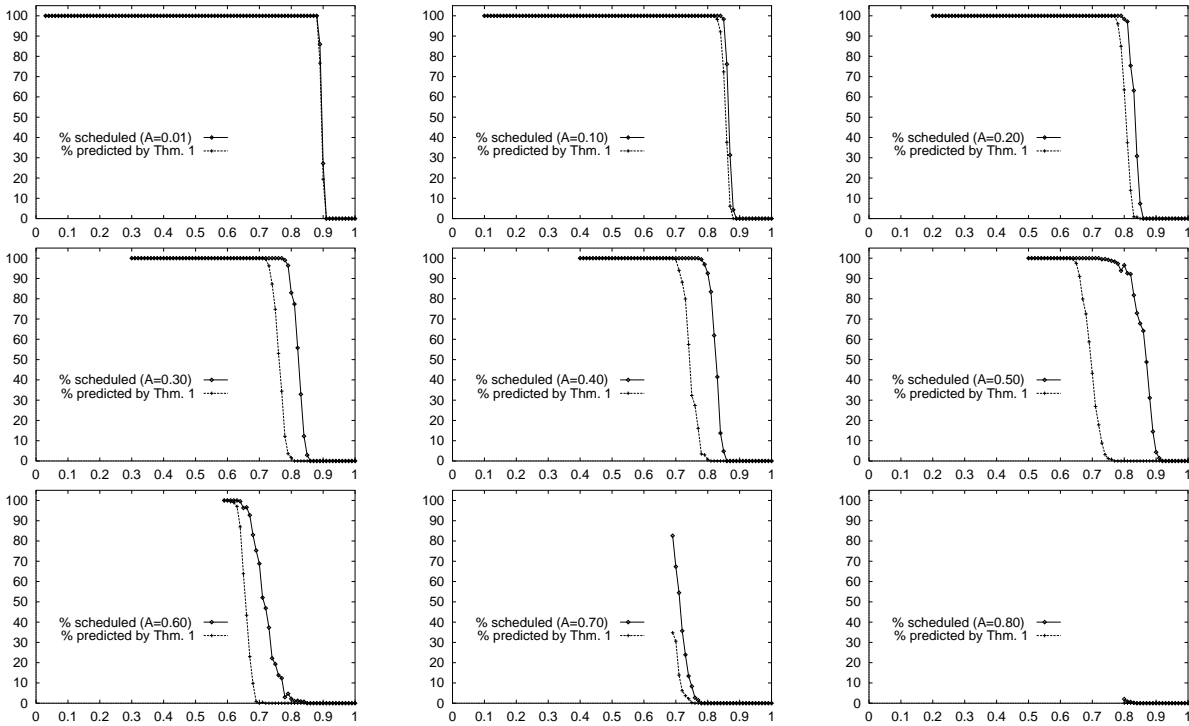


Figure 2. Simulation results for 32 processors,  $F = 0.1$ .

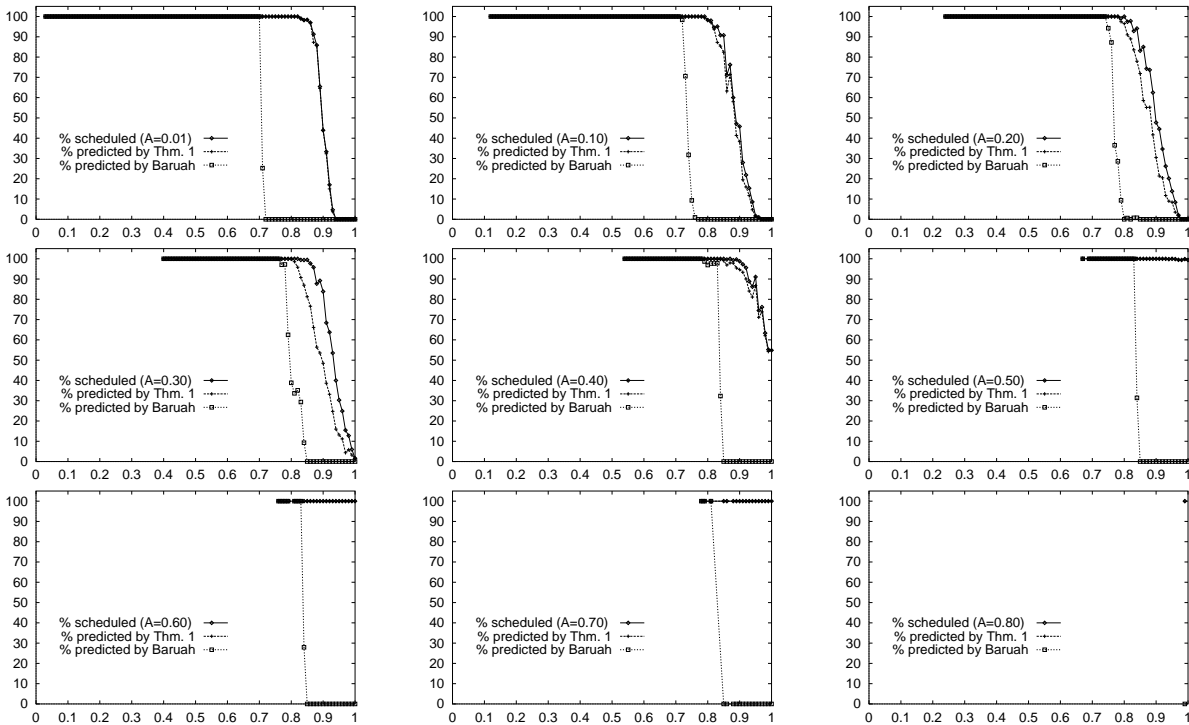
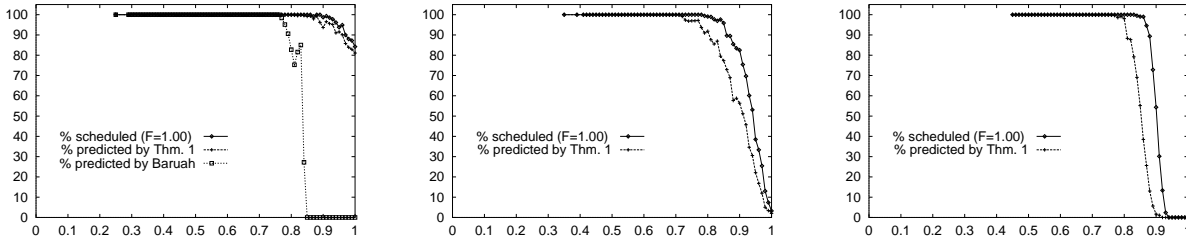


Figure 3. Simulation results for 1 processor,  $F = 0.1$ .



**Figure 4.** Simulation results for 1,2, and 32 processors,  $F = 1$ .

While we have restricted our presentation to the context of scheduling migratable real-time tasks on multiprocessors, our work clearly has implications in other contexts. For example, our ideas could be used to substantially simplify the solution to the Parallel Switching problem [1] by producing a static-priority scheduling scheme and associated schedulability condition for this problem.

The key technique used to achieve our condition was a strong per-task fairness requirement that directly precludes the possibility that a task is scheduled on two processors simultaneously. (Usually such a requirement must be explicitly stated.) This facilitates our relatively straightforward counting argument. We believe this technique will be of value in deriving other multiprocessor scheduling results.

It would be interesting to obtain a tighter condition. However, given the accuracy of the condition presented here, it seems that a tighter condition will yield relatively little improvement in most cases of interest. We would also like to extend our result to allow dynamic arrival and departure of tasks with our condition (or something similar) as an admission control test. As a first step, we have accounted in this paper for tasks with arbitrary release times.

**Acknowledgements:** The authors are grateful to Sanjoy Baruah for valuable feedback and encouragement.

## References

- [1] J. Anderson, S. Baruah, and K. Jeffay. Parallel switching in connection-oriented networks. In *Proceedings of the 20th IEEE Real Time Systems Symposium*, 1999.
- [2] N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, Sept 1993.
- [3] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the 16th Annual IEEE Real-Time Systems Symposium*, pages 200–209, 1995.
- [4] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.
- [5] S. Baruah, J. Gehrke, and G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, 1995.
- [6] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.
- [7] S. Davari and C. L. Liu. An On-Line Algorithm for Real-Time Tasks Allocation. In *Proceedings of the 7th IEEE Real Time Systems Symposium*, pages 194–200, 1986.
- [8] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.
- [9] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1988.
- [10] M. Harbour, M. Klein, and J. Lehoczky. Fixed-priority scheduling of periodic tasks with varying execution priority. In *Proceedings of the 12th IEEE Real Time Systems Symposium*, pages 116–128, 1991.
- [11] A. W. K. Tindell, A. Burns. Mode changes in priority pre-emptively scheduled systems. In *Proceedings of the 13th Annual IEEE Real-Time Systems Symposium*, pages 100–109, 1992.
- [12] S. Lauzac, R. Melhem, and D. Mossé. An efficient RMS admission control and its application to multiprocessor scheduling. In *International Parallel Processing Symposium*, pages 511–518, 1998.
- [13] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real Time Systems Symposium*, pages 201–209, 1990.
- [14] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real Time Systems Symposium*, pages 166–171, 1989.
- [15] J. Y.-T. Leung and J. Whitehead. On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.
- [16] Y. Oh and S. H. Son. Tight performance bounds of heuristics for a real-time scheduling problem. Technical Report CS-93-24, University of Virginia, 1993.
- [17] 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.
- [18] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th Annual IEEE Real-Time Systems Symposium*, pages 288–299, 1996.