Efficient Scheduling of Soft Real-time Applications on Multiprocessors [*]

Anand Srinivasan and James H. Anderson

Department of Computer Science

University of North Carolina at Chapel Hill

E-mail: {anands,anderson}@cs.unc.edu

December 2002

Abstract

In soft real-time applications, tasks are allowed to miss their deadlines. Thus, less-costly scheduling algorithms can be used at the price of occasional violations of timing constraints. This may be acceptable if reasonable tardiness bounds (*i.e.*, bounds on the extent to which deadlines may be missed) can be guaranteed.

In this paper, we consider soft real-time applications implemented on multiprocessors. Pfair scheduling algorithms are the only known means of optimally scheduling *hard* real-time applications on multiprocessors. For this reason, we consider the use of such algorithms here. In the design of Pfair scheduling algorithms, devising schemes to correctly break ties when several tasks have the same deadline is a critical issue. Such tie-breaking schemes entail overhead that may be unacceptable or unnecessary in soft real-time applications. In this paper, we consider the earliest pseudo-deadline first (EPDF) Pfair algorithm, which avoids this overhead by using *no* tie-breaking information. Our main contributions are twofold. First, we establish a condition for ensuring a tardiness of at most one quantum under EPDF. This condition is very liberal and should often hold in practice. Second, we present simulation results involving randomly-generated task sets, including those that do not satisfy our condition. In these experiments, deadline misses rarely occurred, and *no* misses by more than one quantum ever occurred.

[*]Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0204312.

1 Introduction

Real-time systems are typically categorized as being either *hard* or *soft*. In a hard real-time system, task deadlines must be guaranteed, while in a soft real-time system, some task deadlines may be missed. Examples of hard real-time systems include fly-by-wire controllers for airplanes, monitoring systems for nuclear reactors, and automotive braking systems. Examples of soft real-time systems include multimedia and gaming systems.

While deadline misses are tolerated in soft real-time systems, they are obviously undesirable, and system quality and performance may be negatively impacted if tasks miss their deadlines either too often or by too much. One criterion used to measure system quality in the study of soft real-time scheduling algorithms is *tardiness*. If a job (*i.e.*, task invocation) with a deadline at time d completes at time t, then its tardiness is $max(0, t-d)$. That is, if a job misses its deadline, then its tardiness indicates by how much.

In this paper, we consider the problem of scheduling soft real-time applications implemented on tightly-coupled multiprocessors, where minimizing tardiness is the main concern. Our specific focus is fair scheduling algorithms based on Pfairness [7]. Such algorithms are the only known way of optimally scheduling recurrent *hard* real-time tasks on multiprocessors [3, 7, 8, 24]. In addition, fairness is interesting in its own right because it results in *temporal isolation*, *i.e.*, each task's processor share is guaranteed even if other tasks "misbehave" by attempting to execute for more than their prescribed shares. Our main contribution is to show that Pfair scheduling algorithms that are optimal for hard real-time systems can be substantially simplified if deadline misses can be tolerated.

Such simplified algorithms may be useful in a number of contexts. For example, consider a web server that services a large number of connections concurrently, some of which involve audio or video streaming that requires quality-of-service (QoS) guarantees. Such guarantees can be ensured by using fair scheduling disciplines. To handle a large number of connections, a multiprocessor platform may be necessary. Ensim Corp. has deployed fair multiprocessor scheduling algorithms in its product line for this very reason [12, 13]. (The algorithms employed by them use heuristics for which tardiness bounds have not been derived.)

The necessity of fair scheduling on multiple resources may also arise in the area of networking. One way to increase network bandwidth is to install multiple parallel links between pairs of connected routers. The problem of scheduling packets on outgoing links at a router then becomes a multiprocessor scheduling problem [1]. Fairness is desirable in this setting, just as it is in single-link scheduling [10, 15, 16, 21, 29]. Similar scheduling issues also arise in optical networks where wavelength-division-multiplexing (WDM) techniques are used to transmit multiple "light packets" at different wavelengths simultaneously [6].

As a final example, consider *multiprocessor* router platforms that process multiple packets simultaneously. The need for multiprocessor platforms in this context is necessitated by the growing disparity between link capacities and processor speeds [11, 14]. Routers built using programmable network processors are destined to implement fairly complex packet-processing functions in software, making *processing* capacity (as opposed to *link* capacity) a critical resource to be managed [26]. In this setting, fair scheduling is needed to ensure that QoS guarantees can be provided to different flows.

Note that in each of the above applications, an extreme degree of fairness that requires *all* deadlines to be met is not warranted. That is, these systems fall within the class of soft real-time applications. Having motivated the usefulness of efficient fair multiprocessor scheduling algorithms for soft real-time systems, we now describe the fair scheduling concepts used in this paper in greater detail. (Formal definitions are given in Sec. 2.) After that, we present a more detailed overview of the contributions of this paper.

Pfair scheduling. Periodic task systems can be optimally scheduled on multiprocessors using *Pfair* scheduling algorithms [3, 5, 7, 8]. Under Pfair scheduling, each task must execute at a uniform rate, while respecting a fixed-size allocation quantum. Uniform rates are ensured by requiring the allocation error for each task to be always less than one quantum, where "error" is determined by comparing to an ideal fluid system. Due to this requirement, each task T is effectively subdivided into quantum-length *subtasks* $T_1, T_2, \ldots$ that must execute within *windows* of approximately equal lengths: if a subtask of a task T executes outside of its window, then T's error bounds are exceeded. The length and alignment of a task's windows is determined by its *weight* or *utilization*, which is the ratio of its per-job execution cost and period. A task's weight determines the processor share it requires. Fig. 1(a) illustrates the subtasks and windows for the first job of a periodic task of weight 8/11. The end of a subtask's window defines a *pseudo-deadline* for that task; for example, in Fig. 1(a), subtask T_2 has a pseudo-deadline at time 3. (Fig. 1(b) gives an example of an *intra-sporadic* task, a notion generalizing the concept of a periodic task that is defined later in Sec. 3.)

At present, three optimal Pfair scheduling algorithms are known: PF [7], PD [8], and PD^2 [3]. These algorithms prioritize subtasks on an earliest-pseudo-deadline-first (EPDF) basis, but differ in the choice of tie-breaking rules. PD^2 is the most efficient of the three and uses two tie-break parameters. In hard real-time systems, the choice of tie-breaking rules is crucial. In particular, if either of the PD^2 tie-breaks is eliminated, then task sets exist in which deadlines are missed [2].

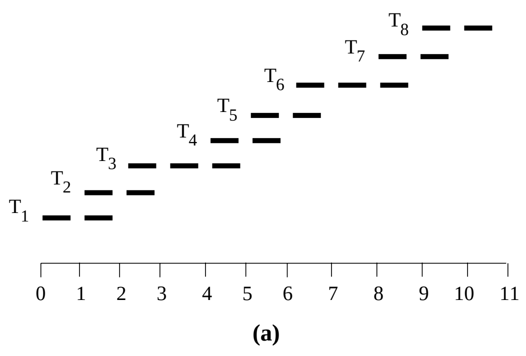
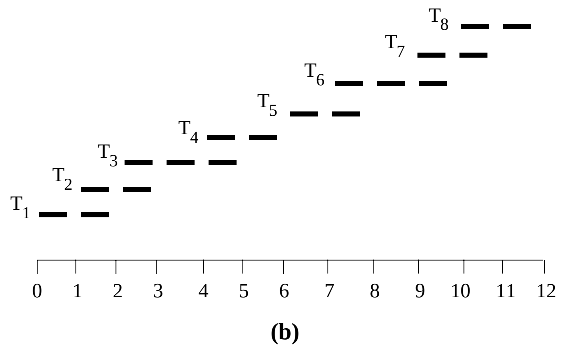


Figure 1: **(a)** Windows of the first job of a periodic task T with weight 8/11. This job consists of subtasks $T_1, \ldots, T_8$, each of which must be scheduled within its window, or else a lag-bound violation will result. (This pattern repeats for every job.) **(b)** The Pfair windows of an IS task. Subtask T_5 becomes eligible one time unit late.

Contributions. In this paper, we consider the following question: Under the simpler EPDF algorithm, in which no tie-break parameters are used, by how much can deadlines be missed? If deadline misses are rare under EPDF, then it may be preferable to use EPDF in soft real-time systems than the more-costly ones noted above. Nonetheless, it is important to note that the two tie-break parameters of PD^2 can be calculated and maintained with only an $O(1)$ cost. Still, using EPDF may be a much better choice. Consider the following.

- For good throughput, a router must process each packet very quickly (typically within 10 ns. in today's technology). Moreover, the number of bits that can be allocated for storing priority information in a packet header may be very limited. (Such information can be sent in packet headers to avoid storing per-flow information in routers [19, 28].) Thus, incurring an additional constant overhead to store and evaluate tie-breaking information may be costly and unnecessary.

- In prior work on fairness in networks [10, 15, 16, 21], much work was done on mechanisms for reallocating spare bandwidth as connections are created and destroyed. In Pfair terminology, this amounts to *reweighting* tasks so that all processing capacity is utilized as tasks dynamically join and leave the system. As explained later, it is possible to reweight a task so that its next pseudo-deadline is preserved. However, weight changes can cause tie-breaking information to change. In the worst case, this may necessitate a complete resorting of the scheduler's priority queue at a $\Theta(N \log N)$ cost, where N is the number of tasks (connections). In a networking application, this cost might be incurred *every time a connection is created or destroyed*. If no tie-breaking information is maintained, this extra overhead can be eliminated.

The main contributions of this paper are twofold. First, we establish a condition for ensuring a tardiness of at most one quantum under EPDF. This condition is very liberal and should often hold in practice. We also generalize this condition for larger tardiness thresholds. Second, we present simulation results involving randomly-generated task sets, including those that do not satisfy our condition (for any tardiness threshold). In these experiments, deadline misses rarely occurred, and *no* misses by more than one quantum ever occurred.

The rest of the paper is organized as follows. In Sec. 2, we give relevant definitions related to Pfair scheduling, and in Sec. 3, we describe the different task models considered in this paper. In Sec. 4, the tardiness bounds mentioned above are derived. Simulation results are presented in Sec. 5. We conclude in Sec. 6.

2 Pfair Scheduling

In defining notions relevant to Pfair scheduling, we limit attention (for now) to periodic tasks.[1] A periodic task T with an integer *period* $T.p$ and an integer *execution cost* $T.e$ has a *weight* $wt(T) = T.e/T.p$, where $0 < wt(T) \leq 1$. A task is called *light* if its weight is less than 1/2, and *heavy* otherwise.

Pfair algorithms allocate processor time in discrete quanta; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. (Hence, *time* t refers to the beginning of slot t.) A task may be allocated time on different processors, but not in the same slot (*i.e.*, interprocessor migration is allowed but parallelism is not permitted). The sequence of allocation decisions over time defines a *schedule* S. Formally, $S\colon \tau \times \mathcal{N} \mapsto \{0, 1\}$, where τ is a set of tasks and $\mathcal{N}$ is the set of nonnegative integers. $S(T, t) = 1$ iff T is scheduled in slot t. In any schedule for M processors, $\sum_{T \in \tau} S(T, t) \leq M$ holds for all t.

Lags and subtasks. The notion of a Pfair schedule is defined by comparing such a schedule to an ideal fluid schedule, which allocates $wt(T)$ processor time to task T in each slot. Deviance from the fluid schedule is formally captured by the concept of *lag*. Formally, the *lag of task T at time t* is[2] $\mathit{lag}(T, t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u)$. T is said to be *over-allocated* at time t if $\mathit{lag}(T, t) < 0$, and *under-allocated* if $\mathit{lag}(T, t) > 0$. A schedule is defined to be *Pfair* iff

$$(\forall T, t :: -1 < \mathit{lag}(T, t) < 1). \qquad (1)$$

Informally, the allocation error associated with each task must always be less than one quantum.

These lag bounds have the effect of breaking each task T into an infinite sequence of *quantum-length subtasks*. We denote the i^{th} subtask of task T as T_i, where $i \geq 1$. As in [7], we associate a *pseudo-release* $r(T_i)$ and *pseudo-*

[1]Unless specified otherwise, we assume that each periodic task begins execution at time 0.
[2]For conciseness, we leave the schedule implicit and use $\mathit{lag}(T, t)$ instead of $\mathit{lag}(T, t, S)$.

deadline[3] $d(T_i)$ with each subtask T_i, as follows. (For brevity, we often drop the prefix "pseudo-.")

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \qquad d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \tag{2}$$

To satisfy (1), T_i must be scheduled in the interval $w(T_i) = [r(T_i), d(T_i))$, termed its *window*. Note that $r(T_{i+1})$ is either $d(T_i) - 1$ or $d(T_i)$. Thus, consecutive windows either overlap by one slot or are disjoint. The *length* of T_i's window, denoted $|w(T_i)|$, is $d(T_i) - r(T_i)$. As an example, consider subtask T_1 in Fig. 1(a). Here, we have $r(T_1) = 0$, $d(T_1) = 2$ and $|w(T_1)| = 2$. Therefore, T_1 must be scheduled at either time 0 or time 1.

Pfair scheduling algorithms. In earlier work [4], we proved that the earliest-pseudo-deadline-first (EPDF) Pfair algorithm is optimal on one or two processors, but not on more than two processors. As its name suggests, EPDF gives higher priority to subtasks with earlier deadlines. A tie between subtasks with equal deadlines is broken arbitrarily. At present, three Pfair scheduling algorithms are known to be optimal on an arbitrary number of processors: PF [7], PD [8], and PD^2 [3]. These algorithms prioritize subtasks on an EPDF basis, but differ in the choice of tie-breaking rules. Selecting appropriate tie-breaks turns out to be *the* most important concern in designing optimal Pfair algorithms. In this paper, our focus is on EPDF, and hence, we do not describe the tie-break parameters of PF, PD, or PD^2.

3 Task Models

In this paper, we mainly consider the *intra-sporadic* (IS) task model proposed by us in earlier work [4, 24] because it provides a general notion of recurrent execution that subsumes that found in the well-studied periodic and sporadic models. The IS model generalizes the sporadic model. The sporadic model generalizes the periodic model by allowing *jobs* to be released "late"; the IS model allows *subtasks* to be released late, as illustrated in Fig. 1(b). More specifically, the separation between $r(T_i)$ and $r(T_{i+1})$ is allowed to be more than $\lfloor i/wt(T) \rfloor - \lfloor (i-1)/wt(T) \rfloor$, which would be the separation if T were periodic. Thus, an IS task is obtained by allowing a task's windows to be right-shifted from where they would appear if the task were periodic. Fig. 1(b) illustrates this.

Let $\theta(T_i)$ denote the offset of subtask T_i, *i.e.*, the amount by which $w(T_i)$ has been right-shifted. Then, by

[3]In our earlier work [3, 4, 24], pseudo-deadlines were defined to refer to slots; here, they refer to time. Hence, the formula for $d(T_i)$ given here is slightly different.

(2), we have the following.

$$r(T_i) = \theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{3}$$

$$d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil \tag{4}$$

The offsets are constrained so that the separation between any pair of subtask releases is at least the separation between those releases if the task were periodic. Formally, the offsets satisfy the following property.

$$k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i) \tag{5}$$

Each subtask T_i has an additional parameter $e(T_i)$ that specifies the first time slot in which it is eligible to be scheduled. It is assumed that $e(T_i) \leq r(T_i)$ and $e(T_i) \leq e(T_{i+1})$ for all $i \geq 1$. Allowing $e(T_i)$ to be less than $r(T_i)$ is equivalent to allowing "early" subtask releases as in ERfair scheduling [3]. The interval $[r(T_i), d(T_i))$ is called the *PF-window* of T_i and the interval $[e(T_i), d(T_i))$ is called the *IS-window* of T_i. (Note that the notion of a job is not mentioned here. For systems in which subtasks are grouped into jobs that are released in sequence, the definition of e would preclude a subtask from becoming eligible before the beginning of its job.)

The IS model is more suitable than the periodic model for the networking examples described in Sec. 1. Due to network congestion and other factors, packets may arrive late or in bursts. The IS model treats these possibilities as first-class concepts and handles them more seamlessly. In particular, a late packet arrival corresponds to an IS delay. On the other hand, if a packet arrives early (as part of a bursty sequence), then its eligibility time will be less than its Pfair release time. Note that its Pfair release time determines its deadline. Thus, in effect, an early packet arrival is handled by postponing its deadline to where it would have been had the packet arrived on time. This is very similar to the approach taken in the (uniprocessor) virtual-clock scheduling scheme [29].

Feasibility. In [4], we showed that an IS task system τ is feasible on M processors iff

$$\sum_{T \in \tau} wt(T) \leq M. \tag{6}$$

In [24], we also proved that PD^2 correctly schedules any feasible IS task system, and thus, is optimal for scheduling IS tasks. In recent work [23], we showed that EPDF is optimal for scheduling IS tasks on M (> 1) processors if the weight of each task is at most $\frac{1}{M-1}$. (It is easy to show that this result is fairly tight. In particular, if the weight of a task is allowed to be at least $\frac{1}{M-1} + \frac{1}{(M-1)^2}$, then EPDF can miss deadlines.)

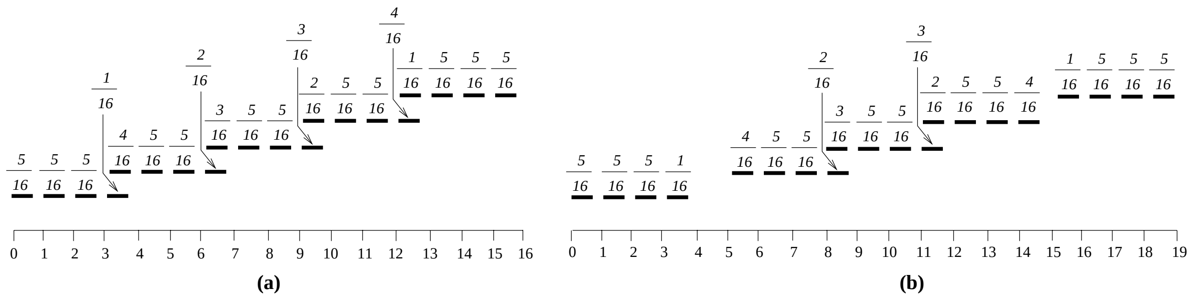


Figure 2: Fluid schedule for the first five subtasks ($T_1, \ldots, T_5$) of a task T of weight 5/16. The share of each subtask in each slot of its PF-window is shown. In **(a)**, no subtask is released late; in **(b)**, T_2 and T_5 are released late. Note that $share(T, 3)$ is either 5/16 or 1/16 depending on when subtask T_2 is released.

Shares and lags in IS task systems. Let v be the time at which an IS task T releases its first subtask. The lag of T at time t is defined in the same way as it is defined for periodic tasks. Let $ideal(T, v, t)$ denote the share that T receives in an ideal fluid (processor-sharing) schedule in $[v, t)$. Then,

$$lag(T, t) = ideal(T, v, t) - \sum_{u=v}^{t-1} S(T, u). \tag{7}$$

Before defining $ideal(T, v, t)$, we define $share(T, u)$, which is the share assigned to task T in slot u. $share(T, u)$ is defined in terms of a function f that indicates the share assigned to each subtask in each slot.

$$f(T_i, u) = \begin{cases} (\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1) \times wt(T) - (i-1), & \text{if } u = r(T_i) \\ i - (\left\lceil \frac{i}{wt(T)} \right\rceil - 1) \times wt(T), & \text{if } u = d(T_i) - 1 \\ wt(T), & \text{if } r(T_i) < u < d(T_i) - 1 \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

Fig. 2 shows the values of f for different subtasks of a task of weight 5/16. $share(T, u)$ is simply defined as $share(T, u) = \sum_i f(T_i, u)$. Observe that $share(T, u)$ usually equals $wt(T)$, but in certain slots, it may be less than $wt(T)$, so that each subtask of T has a unit share. Using (8), it is easy to show the following.

(F1) For all time slots t, $share(T, t) \leq wt(T)$.

We can now define $ideal(T, v, t)$ as $\sum_{u=v}^{t-1} share(T, u)$. Hence, from (7), $lag(T, t+1) = \sum_{u=0}^{t}(share(T, u) - S(T, u)) = lag(T, t) + share(T, t) - S(T, t)$. Similarly, the total lag for a schedule S and task system τ at time $t + 1$, denoted by $LAG(\tau, t + 1)$, is defined as follows. ($LAG(\tau, 0)$ is defined to be 0.)

$$LAG(\tau, t+1) = LAG(\tau, t) + \sum_{T \in \tau}(share(T, t) - S(T, t)). \tag{9}$$

3.1 Dynamic Tasks

In each of the examples considered in Sec. 1, connections are not permanent. Thus, in addition to mechanisms for handling burstiness and delays, support for task dynamism is needed. When considering dynamic task systems, the key issue is that of devising conditions under which tasks may join and leave the system.

A condition for joining is an immediate consequence of the feasibility test in (6), *i.e.*, admit a task if the total utilization is at most M after its admission. The important question left is: *when should a task be allowed to leave the system*? (Here, we are referring to the time when the task's utilization can be reclaimed. The task may actually be allowed to leave the system earlier.) As shown in [9, 27], if an over-allocated task (a task with negative lag) is allowed to leave, then it can re-join immediately and effectively execute at a rate higher than its specified rate causing other tasks to miss their deadlines. It is easy to show that, in such a case, tardiness cannot be bounded by any constant value even on one processor. Therefore, as in [9, 27], we allow only non-over-allocated tasks (*i.e.*, tasks with non-negative lags) to leave the system, as stated below.

(J) *Join condition*: A task T can join at time t iff the total utilization after joining is at most M. If T joins at time t, then $\theta(T_1)$ is set to t. (A task that re-joins after having left can be viewed as a new task.)

(L) *Leave condition*: A task T can leave at time t iff $t \geq d(T_i)$, where T_i is the last-released subtask of T.

The condition $t \geq d(T_i)$ implies that $lag(T, t) \geq 0$. To see why, note that since T_i is the last-released subtask of T, by time $d(T_i)$ it receives a share equal to i in the ideal schedule. In the EPDF schedule, it cannot have received more than i time units. Thus, $lag(T, t) \geq 0$.

Actually, assuming that tasks leave only with zero lag is consistent with Condition (L), so we assume this in our proofs (Sec. 4). To see why this is reasonable, suppose that T leaves with positive lag; this implies that it has released more subtasks than have been scheduled. Let T_i be T's last-*released* subtask and T_k $(k < i)$ be its last-*scheduled* subtask. Since T leaves at time t, $T_{k+1}, \ldots, T_i$ do not need to be scheduled; thus, we can assume they were never released. In that case, the share in the ideal schedule for T equals k, and hence, $lag(T, t) = 0$.

Partitioning versus Pfair scheduling. One commonly-used approach in multiprocessor scheduling is partitioning, in which each task is statically assigned to a particular processor. Unfortunately, finding an optimal assignment of tasks to processors is NP-hard and non-optimal heuristics need to be used. Another problem with partitioning is that it is inherently sub-optimal: task sets with utilization at most M cannot always be partitioned. Nonetheless, partitioning has its advantages: migration overhead is zero and simpler and widely-studied uniprocessor scheduling algorithms can be used on each processor. However, partitioning is quite problematic if task dynamism is allowed. In particular, every new task that joins can potentially cause a re-partitioning of the entire system. The resulting overhead would clearly be unacceptable in many applications.

While the frequency of preemptions under Pfair scheduling may be a potential concern, a recent experimental comparison conducted by us and colleagues [25] showed that PD^2 has comparable performance (in terms of schedulability) to earliest-deadline-first (EDF) scheduling with partitioning. In this study, real system overheads such as context-switching costs were considered. Moreover, the study was biased *against* Pfair scheduling in that only static systems with independent[4] tasks of low utilization[5] were considered. In short, PD^2 and multiprocessor-EDF were comparable because the schedulability loss due to migration and preemption costs under PD^2 was comparable to the loss in schedulability due to partitioning under EDF.

3.2 Task Reweighting

As mentioned in Sec. 1, task reweighting might be necessary to fully utilize all processors as tasks join and leave the system. A task can be reweighted by allowing it to leave with its old weight and re-join with its new weight. In EPDF-scheduled systems, this reweighting can be done in a simple manner using the following procedure, which ensures that the relative priorities of the current subtasks do not change. This procedure has the significant advantage that the scheduler's current priority queue of eligible subtasks does not have to be resorted when reweighting occurs. Suppose that task T needs to be reweighted at time t. Let T_i be the subtask of T that is eligible at t. Task T can be reweighted by simply replacing it by a new task U with the new weight and by aligning U_1's window so that $d(U_1) = d(T_i)$ and $e(U_1) \leq t$. (In reality, T's weight can simply be redefined, instead of creating a new task.) This procedure is problematic under PF, PD, or PD^2 because tie-break parameters also have to be matched to avoid changing task priorities, and this is not always possible.

Unfortunately, the reweighting procedure described above complicates the definition of the ideal system, and we do not have sufficient space to describe the changes needed. Thus, in the next section, we assume that this procedure is not used. However, the results proved there do hold if this procedure is used.

[4]Considering only independent tasks is advantageous to EDF. While efficient synchronization techniques for Pfair-scheduled systems exist [18], to the best of our knowledge, no general synchronization protocols that are directly applicable to multiprocessor-EDF have been proposed in the literature. While the multiprocessor priority ceiling protocol [22] probably could be adapted for this purpose, its overhead in EDF-scheduled systems would likely be quite high.

[5]The performance of partitioning approaches is relatively good when individual task utilizations are low. As the mean task utilization increases, the schedulability loss due to partitioning increases. Note that tasks with high utilization may arise in systems that are hierarchically scheduled [17, 20].

4 Tardiness Bounds for EPDF

The notion of tardiness for jobs was defined earlier in Sec. 1. *Tardiness for subtasks* is defined similarly: if t is the time at which subtask T_i completes, then $max(0, t - d(T_i))$ is its tardiness. The *tardiness of a task system* is defined as the maximum tardiness among all of its subtasks in any schedule. In this section, we present a condition on task weights that is sufficient for ensuring that EPDF maintains a tardiness of at most one quantum. Later, we generalize this condition for tardiness thresholds that exceed one.

Before continuing, we introduce some notation to refer to slots in which some processors are idle. In a schedule S, if k processors are idle in slot t, then we say that there are k *holes* in S in slot t. Note that holes may exist because of late subtask releases, even if total utilization is M. The lemma below concerning LAG values and holes is used in our proofs. It is proved in [24].

Lemma 1 *If $LAG(\tau, t) < LAG(\tau, t+1)$, then there is a hole in slot t.*

Proof Sketch: Suppose there is no hole in slot t. Then, the total share in the EPDF schedule increases by M from t to $t+1$. On other hand, the share in the ideal schedule can increase by at most M, since the total weight of all tasks at any time is at most M (by (J)). Thus, LAG cannot increase from t to $t+1$. □

4.1 Sufficient Condition for Tardiness of at most One

We prove that any feasible task system that satisfies the following condition has a tardiness of at most one.

(M1) At all times, the sum of the $M-1$ largest task weights is at most $\frac{M+1}{2}$.

Assume to the contrary that there exists a feasible task system τ satisfying (M1) with tardiness greater than one. Let T_i be the subtask (in some given schedule) with the earliest deadline among all subtasks with tardiness greater than one, and let $t_d = d(T_i)$. Thus, all subtasks with deadlines less than t_d have tardiness at most one.

It is easy to see that the tardiness of T_i is not affected by subtasks with deadlines greater than t_d. Note that any such subtask is scheduled before t_d only if no subtask with a deadline at most t_d is eligible at that slot. In other words, the presence or absence of subtasks with deadlines beyond t_d does not change the scheduling of subtasks with deadlines at most t_d. Thus, we can assume that no task in τ releases any subtask with a deadline greater than t_d. In other words,

$$\text{for every subtask } U_j \in \tau, d(U_j) \leq t_d. \tag{10}$$

We first show that for T_i to have a tardiness of at least two, at least $M+1$ subtasks miss their deadlines at t_d. This, in turn, implies that the LAG of τ at time t_d is at least $M+1$.

Lemma 2 *At least $M+1$ subtasks in τ miss their deadlines at t_d.*

Proof: Consider any subtask U_j such that U_j misses its deadline at t_d. Because $d(U_j) = t_d$, by (4) and (5), the deadline of U_{j-1} is at or before $t_d - 1$. Therefore, by the definition of T_i and t_d, U_{j-1} has tardiness at most one and is scheduled in $[0, t_d)$. Hence, U_j is eligible to be scheduled at time t_d. If there are at most M subtasks like U_j (including T_i), then each of these subtasks will be scheduled in $[t_d, t_d + 1)$. Therefore, subtask T_i cannot have tardiness greater than one. Contradiction. □

Lemma 3 $LAG(\tau, t_d) \geq M + 1$.

Proof: By (9), we have

$$LAG(\tau, t_d) = \sum_{t=0}^{t_d-1} \sum_{T \in \tau} share(T, t) - \sum_{t=0}^{t_d-1} \sum_{T \in \tau} S(T, t).$$

The first term on the right-hand side of the above equation is the total share in the ideal schedule in $[0, t_d)$, which equals the total number of subtasks in τ. (This follows from (10) and because τ is feasible.) The second term corresponds to the number of subtasks scheduled by EPDF in $[0, t_d)$. Since at least $M+1$ subtasks miss their deadlines at t_d (by Lemma 2), the difference between these two terms is at least M+1. □

Because $LAG(\tau, 0) = 0$, it follows by Lemma 3 that there exists a time $t < t_d$ such that $LAG(\tau, t) < M + 1$ and $LAG(\tau, t+1) \geq M + 1$. We now prove some properties about task lags at time $t+1$; using these properties and (M1), we later derive a contradiction concerning the existence of t.

By Lemma 1, there is at least one hole in slot t (*i.e.*, in $[t, t+1)$). Therefore, if A denotes the set of tasks scheduled in slot t, then we have

$$|A| \leq M - 1. \tag{11}$$

Let B denote the set of tasks not in A that are "active" at t. A task U is *active* at time t if it has a subtask U_j such that $e(U_j) \leq t < d(U_j)$. (A task may be inactive either because it has already left the system or because of a late subtask release.) Consider any task $U \in B$ and let U_j be such that $e(U_j) \leq t < d(U_j)$. Because slot t has a hole and no subtask of U is scheduled at t, and since $e(U_j) \leq t < d(U_j)$, U_j must be scheduled in $[0, t)$.

Let I denote the set of the remaining tasks that are not active at time t. Fig. 3(a) shows how the tasks in A, B, and I are scheduled. We now estimate the *lag* values for the tasks in each of A, B, and I at time $t+1$.

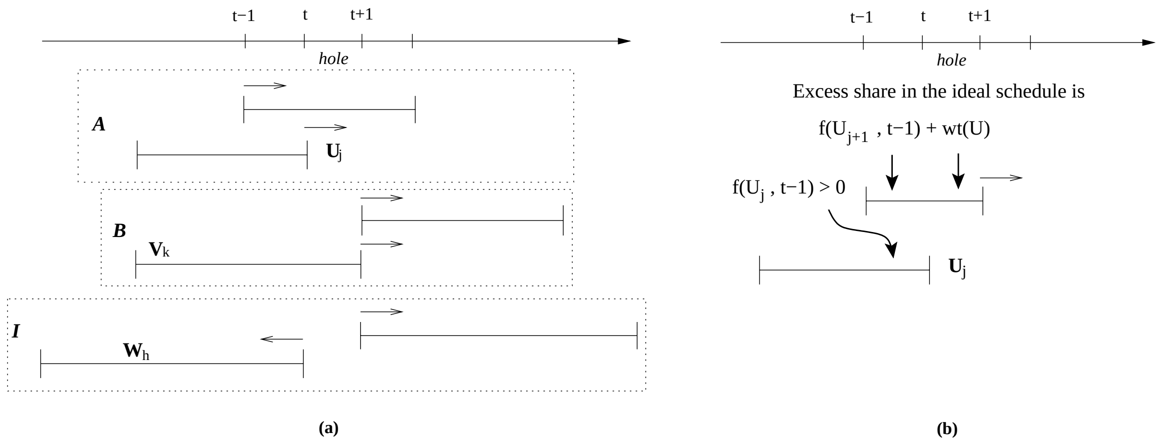


Figure 3: In this figure, PF-windows are denoted by line segments. An arrow over a release (respectively, deadline) indicates that the release (respectively, deadline) could be anywhere in the direction of the arrow. There is a hole in slot t. **(a)** Sets A, B, and I. The PF-windows of a sample task of each set are shown. **(b)** Case 4 of Lemma 6. The excess share received by U in $[0, t+1)$ in the ideal schedule is equal to the shares received by subtasks after U_j in $[0, t+1)$. (Note that if U_{j+1} has a deadline at time $t+1$, then the PF-window of U_{j+2} may begin at time t, in which case part of the $wt(U)$ share in slot t is due to U_{j+2}.)

Lemma 4 *For $W \in I$, $lag(W, t+1) = 0$.*

Proof: Consider any subtask W_h of task W. If $e(W_h) \geq t+1$, then $r(W_h) \geq t+1$. Therefore, by (8), $f(W_h, u) = 0$ for all slots $u \leq t < r(W_h)$. Hence, the share of W_h in the ideal schedule in $[0, t+1)$ is zero. Also, in the EPDF schedule, W_h is scheduled at or after $t+1$. On the other hand, if $e(W_h) \leq t$, then by the definition of I, $d(W_h) \leq t < t_d$. Because the tardiness of such a subtask is at most one, W_h is scheduled in $[0, t+1)$. Hence, the share received by W_h in $[0, t+1)$ is one in both the ideal and EPDF schedules. Thus, for all subtasks of W, the share of that subtask in $[0, t+1)$ is the same in both the ideal and EPDF schedules. (Furthermore, recall from Sec. 3.1 that any task that leaves the system does so with zero lag.) Therefore, $lag(W, t+1) = 0$. □

Lemma 5 *For $V \in B$, $lag(V, t+1) \leq 0$.*

Proof: Consider any subtask V_k of task V. Again, as in the proof of Lemma 4, if $r(V_k) \geq t+1$, then by (8), the share of V_k in $[0, t+1)$ in the ideal schedule is zero. On the other hand, if $r(V_k) \leq t$, then, as discussed earlier, V_k is scheduled before t because of the hole in slot t. Thus, the share of V_k in $[0, t+1)$ is one in the EPDF schedule, and at most one in the ideal schedule. ($d(V_k)$ may be greater than $t+1$, in which case a portion of V_k's share in the ideal schedule is allocated after $t+1$.) Thus, for any subtask of V, its share in $[0, t+1)$ in the EPDF schedule is at least its share in $[0, t+1)$ in the ideal schedule. Hence, $lag(V, t+1) \leq 0$. □

Lemma 6 *For $U \in A$, $lag(U, t+1) < 2 \times wt(U)$.*

Proof: Let U_j be the subtask of U scheduled at time t. If $d(U_j) \leq t$, then because $t < t_d$, $d(U_j) < t_d$. By the choice of T_i and t_d, it follows that the tardiness of U_j is at most one. Therefore, $d(U_j) = t$ in this case. Thus, in general, $d(U_j) \geq t$. We now consider four cases depending on the value of $d(U_j)$.

Case 1: $\boldsymbol{d(U_j) > t+1}$. In this case, by (3)–(5), $r(U_{j+1}) \geq t+1$. Reasoning exactly as in the proof of Lemma 5, we can show that $lag(U, t+1) \leq 0$.

Case 2: $\boldsymbol{d(U_j) = t+1 \wedge b(U_j) = 0}$. Again, by (3)–(5), $r(U_{j+1}) \geq t+1$. It follows that the share received by any subtask U_k $(k > j)$ in the ideal schedule in $[0, t+1)$ is zero. Thus, the share received by task U in $[0, t+1)$ is the same in both the ideal and EDPF schedules. Therefore, $lag(U, t+1) = 0$.

Case 3: $\boldsymbol{d(U_j) = t+1 \wedge b(U_j) = 1}$. By (3)–(5), we have

$$r(U_{j+1}) \geq d(U_j) - 1. \tag{12}$$

Therefore, $r(U_{j+1}) \geq t$. Similarly, by (3)–(5), we obtain $r(U_{j+2}) \geq t+1$. Therefore, the share of any subtask U_k $(k > j+1)$ is zero in $[0, t+1)$ in the ideal schedule.

Now, the total share that U receives in $[0, t+1)$ in the EPDF schedule is j. However, in the ideal schedule, the share received by U in $[0, t+1)$ may be more because of the share that U_{j+1} (if it exists) receives in $[0, t+1)$. This share will be non-zero only if $r(U_{j+1}) \leq t$, *i.e.*, $r(U_{j+1}) \leq d(U_j) - 1$. In this case, by (12), we have $r(U_{j+1}) = d(U_j) - 1$. This implies that $\theta(U_{j+1}) = \theta(U_j)$. It also implies that the excess share in the ideal schedule in $[0, t+1)$ is at most $f(U_{j+1}, t)$. Because $r(U_{j+1}) = t$, by (8), we have $f(U_{j+1}, t) = (\lfloor j/wt(U) \rfloor + 1) \cdot wt(U) - j$.

Because $\theta(U_{j+1}) = \theta(U_j)$, by (3) and (4), $\lfloor j/wt(U) \rfloor = \lceil j/wt(U) \rceil - 1$. Hence, $\lfloor j/wt(U) \rfloor < j/wt(U)$. Therefore, $f(U_{j+1}, t) < (j/wt(U) + 1) \cdot wt(U) - j$, *i.e.*, $f(U_{j+1}, t) < wt(U)$. Thus, $lag(U, t+1) < wt(U)$.

Case 4: $\boldsymbol{d(U_j) = t}$. In this case, by (3)–(5), we have $r(U_{j+1}) \geq t-1$ and $r(U_{j+2}) \geq t$. Note that all subtasks of U up to and including U_j receive a share of one over $[0, t+1)$ in both the EPDF and ideal schedules (see Fig. 3(a)). By (8), no subtask receives a share in the ideal schedule before its Pfair release time. Therefore, the only subtask after U_j that may contribute to U's share in the ideal schedule in $[0, t)$ is U_{j+1}. Further, by (F1), the share of U in slot t (*i.e.*, in $[t, t+1)$) is at most its weight. Thus, it follows that the excess share that U can receive in $[0, t+1)$ in the ideal schedule is at most $f(U_{j+1}, t-1) + wt(U)$ (refer to Fig. 3(b)).

The first term, $f(U_{j+1}, t-1)$, will be non-zero only if $r(U_{j+1}) = t-1$, *i.e.*, $r(U_{j+1}) = d(U_j)-1$. Reasoning exactly as in Case 3, it follows that $f(U_{j+1}, t-1) < wt(U)$. Hence, $lag(U, t+1) < 2 \times wt(U)$.

Thus, in all cases, we have $lag(U, t+1) < 2 \times wt(U)$. $\square$

Because $LAG(\tau, t+1) = \sum_{U \in A \cup B \cup I} lag(U, t+1)$, by Lemmas 4–6, $LAG(\tau, t+1) < 2 \times \sum_{U \in A} wt(U)$. By (11), $|A| \leq M-1$. Therefore, by (M1), $LAG(\tau, t+1) < M+1$, contradicting our assumption about t. Thus, we have the following theorem.

Theorem 1 *Every feasible IS task system satisfying (M1) has a tardiness of at most one under EPDF.*

Discussion. One way to ensure (M1) is to restrict the weight of every task to at most $\frac{M+1}{2M-2}$. Note that $\frac{M+1}{2M-2} > \frac{1}{2}$. Thus, EPDF guarantees a tardiness of at most one for a task system consisting solely of light tasks.

It is possible to improve (M1) by more accurately bounding the lag values for tasks in set A. In particular, we can show that there must be at least one task in A with a deadline at $t+1$. Thus, either Case 2 or 3 of Lemma 6 applies for that task. It can be shown that the lag for that task at time $t+1$ is bounded by its weight. This gives us the following (slightly) improved condition.

(M1′) Let $w_1, w_2, \ldots, w_{M-1}$ denote the $M-1$ largest task weights in non-increasing order. Then, $w_{M-1} + 2 \times \sum_{i=1}^{M-2} w_i \leq M+1$.

This allows us to improve the individual weight restriction to $\frac{M+1}{2M-3}$. Note that, for $M \leq 4$, we have $2M-3 \leq M+1$, *i.e.*, $\frac{M+1}{2M-3} \geq 1$. Because the weight of any task is at most one, the individual weight restriction is always satisfied if $M \leq 4$. That is, for $M \leq 4$, EPDF guarantees a tardiness of at most one with no weight restriction.

Tightness. At present, we do not know whether the above condition is tight. Though we have not been able to find an M-processor schedule in which more than M subtasks miss their deadlines simultaneously, we do have examples in which up to $M-2$ simultaneous misses occur.

Consider the following set of periodic tasks: three tasks of weight $\frac{1}{2}$ and $M-1$ tasks of weight $\frac{2M-3}{2M-2}$. Fig. 4 illustrates the case $M = 5$. In these examples, the number of simultaneous deadline misses increases up to some multiple of the least common multiple of the task periods, and then remains steady after that. Note that, in these examples, the percentage of jobs that miss their deadlines is quite high (approaches 25% for large M). However, as our simulation experiments (described in Sec. 5) indicate, such cases are very rare indeed.

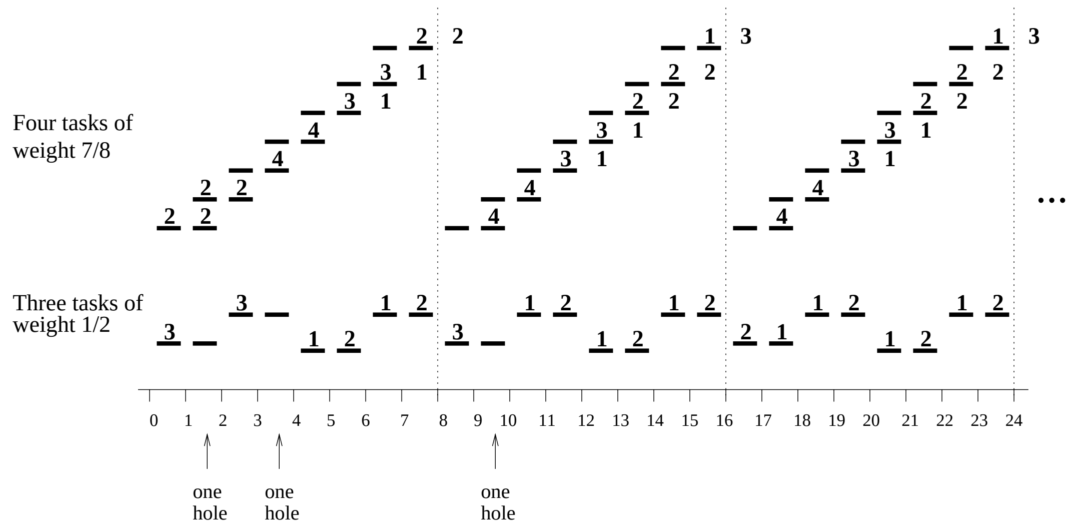


Figure 4: The Pfair window of each subtask is shown on a separate line. An integer value n in slot t means that n of the corresponding subtasks are scheduled in slot t. No integer value means that no such subtask is scheduled in slot t. Each subtask's IS-window is same as its PF-window. Subtasks that miss their deadlines are shown scheduled after their windows. In the schedule shown, EPDF breaks ties in favor of the tasks of weight $1/2$. Note that the schedule over $[16, 24]$ repeats after time 24. Thus, a maximum of three subtasks miss their deadlines simultaneously, and hence, the tardiness of each subtask is at most one.

4.2 Generalizing the Condition

The above approach can be easily extended to obtain conditions similar to (M1) for guaranteeing a tardiness of at most k. In particular, the following condition ensures that tardiness is never more than k.

(Mk) At all times, the sum of the $M-1$ largest task weights is at most $\frac{kM+1}{k+1}$.

The proof remains almost the same, except the lag of a task U in A may be up to $(k+1)$ times the weight of U. Informally, this can happen because the subtask of U scheduled in slot t (which has a hole) may have a deadline at $t-k$ (since, prior to t_d, deadlines can be missed by up to k slots) and hence, its successor may have a release at time $t-k$. Reasoning as in the proof of Lemma 6, we can show that the excess share for U over the interval $[t-k, t+1)$ is less than $(k+1)$ times the weight of U. The required result then follows.

As before, we can improve (Mk) to (Mk$'$) as follows.

(Mk$'$) Let $w_1, w_2, \ldots, w_{M-1}$ denote the $M-1$ largest task weights in non-increasing order. Then, $w_{M-1} + (k+1) \times \sum_{i=1}^{M-2} w_i \le kM+1$.

Condition (Mk) gives us an individual weight restriction of $\frac{kM+1}{(k+1)(M-1)}$, while (Mk$'$) gives us a slightly better value of $\frac{kM+1}{(k+1)(M-2)+1}$. Note that both these values are at least $\frac{k}{k+1}$.

5 Simulation Results

To determine how frequently deadlines are missed under EPDF, and by how much, we computed EPDF schedules for a number of randomly generated task sets. Only periodic (not IS) task sets were considered in these experiments. Intuitively, introducing IS delays should only reduce demand and hence lessen the likelihood of missed deadlines. In addition, each task set was defined to fully utilize all available processors to further increase the possibility of deadline misses.

The following procedure was followed in every run of the simulation. First, the number of processors was chosen as a random number between 1 and 32. Then, task weights were generated randomly in the interval $(0, 1]$. (The weight restrictions of (M1) or (Mk) and their variants were not applied.) The weight of the last task was chosen so that the total utilization equaled the number of processors. For each generated task set, we constructed an EPDF schedule over the time interval $[0, 10L)$, where L is the least common multiple of all task periods. We repeated this procedure approximately 190,000 times. Thus, the number of data points obtained per processor in each graph is approximately 6,000.

In all the runs, no subtask ever missed its deadline by more than one quantum. In other words, the tardiness for each generated task set was at most one. Though this is not a proof, it does strongly indicate that any task set for which tardiness is greater than one is probably pathological and rare, if such a task set exists.

The graph in Fig. 5 plots the percentage of task sets with tardiness greater than zero versus the number of processors. (Recall that a task set has tardiness greater than zero even if just a *single* job misses its deadline.) For example, EPDF misses at least one deadline for 21% of the generated task sets on three processors. (No task set misses a deadline for systems of one or two processors because of the optimality of EPDF for such systems [4].) The percentage of task sets with non-zero tardiness initially increases as the number of processors increases and then steadies. The maximum is around 40%. Though this value may seem high, as the remaining graphs show, the fraction of deadlines that were missed tended to be extremely low.

In Fig. 6, the average percentage of deadlines missed is shown versus the number of processors. Inset (a) shows the average over all generated task sets, while inset (b) shows the average over task sets that have at least one deadline miss. Both the percentage of job deadlines missed and the percentage of subtask deadlines missed are plotted. As can be seen from these graphs, deadline misses are quite rare.

Note that the percentage of job deadlines missed is more than the percentage of subtask deadlines missed. There are two reasons for this. First, the total number of subtask deadlines is much more than the total number

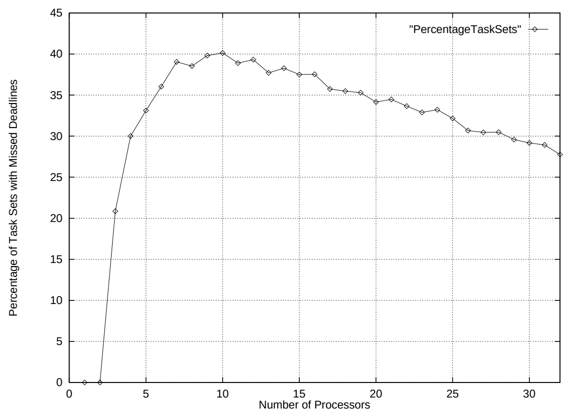
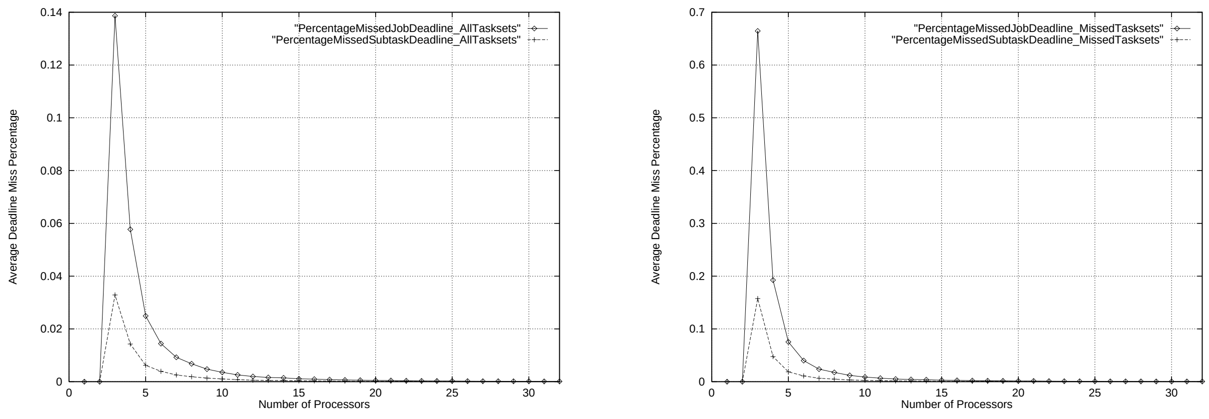


Figure 5: Percentage of task sets with non-zero tardiness versus the number of processors.



(a) **(b)**

Figure 6: Solid (dotted) lines denote the percentage of job (subtask) deadlines missed. **(a)** Percentage of deadlines missed averaged over all task sets. **(b)** Percentage of deadlines missed averaged over task sets with non-zero tardiness.

of job deadlines. Second, subtask deadline misses within a job have a tendency to cascade, causing the later subtasks within a job (and hence the job itself) to have a higher likelihood of missing their deadlines.

Surprisingly, the largest average of job deadlines missed in Fig. 6 is obtained for three processors: 0.14% in inset (a) and 0.67% in inset (b). It is also surprising that the percentage of misses decreases as the number of processors increases. Recall from Sec. 3 that EPDF is optimal on M processors if the weight of each task is at most $\frac{1}{M-1}$. (Also recall that this condition is fairly tight.) As M increases, $\frac{1}{M-1}$ decreases and hence, intuitively, the chance of a deadline miss should also increase.

However, as M increases, more tasks need to be generated to fully utilize the system. Because our samples are generated randomly, the probability of having low-weight tasks in the system also increases. This in turn drives the number of tasks (and hence, the total number of deadlines) up. Thus, though the number of missed deadlines may increase, the percentage of deadline misses decreases. Analogously, for fewer processors, deadline misses are more common when most tasks have a large weight. In this case, the number of tasks is small and therefore, the percentage of deadlines missed tends to be higher.

6 Conclusions

We have considered the scheduling of soft real-time tasks on multiprocessors. We have established a sufficient condition for ensuring that the EPDF algorithm meets a given tardiness threshold. Simulation results indicate that the percentage of deadlines missed is very low even for systems that do not satisfy our condition (for any threshold). These experiments also indicate that the performance (in terms of tardiness and percentage of missed deadlines) of EPDF scales well as the number of processors increase.

We conjecture that no weight restriction is needed to ensure a tardiness threshold of one under EPDF. To date, we (and several of our colleagues) have expended considerable effort (with little success) trying to show this. This problem appears to be of *considerable* difficulty. Though our weight restrictions should often hold in practice, there are cases in which tasks of heavier weight may be necessary. Such cases can arise in hierarchical scheduling approaches, in which several tasks are combined into a single *supertask* for efficiency and other reasons [17, 20]. For such systems, it would be desirable to have no weight restrictions.

We are also currently studying systems that have a mix of hard and soft real-time tasks. For such systems, we would like to show that if only hard real-time tasks use tie-break parameters, then no hard deadlines will be missed, and soft deadlines will be missed by at most one quantum. This also appears to be an issue of considerable difficulty.

References

[1] J. Anderson, S. Baruah, and K. Jeffay. Parallel switching in connection-oriented networks. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 200–209, Dec. 1999.

[2] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Submitted to the Journal of Computer Systems and Sciences.*

[3] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proc. 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 *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, Dec. 2000.

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

[6] E. Bampis and G. N. Rouskas. The scheduling and wavelength assignment problem in optical wdm networks. *IEEE/OSA Journal of Lightwave Technology*, 20(5):782–789, 2002.

[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 *Proc. of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[9] S. Baruah, J. Gehrke, C. G. Plaxton, I. Stoica, H. Abdel-Wahab, and K. Jeffay. Fair on-line scheduling of a dynamic set of tasks on a single resource. *Information Processing Letters*, 26(1):43–51, Jan. 1998.

[10] J. Bennett and H. Zhang. WF^{2}Q: Worst-case fair queueing. In *Proc. of IEEE INFOCOM*, pages 120–128, March 1996.

[11] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture*, June 2000.

[12] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proc. of the Fourth ACM Symposium on Operating System Design and Implementation*, pages 45–58, Oct. 2000.

[13] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proc. of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 3–14, May 2001.

[14] K. Coffman and A. Odlyzko. The size and growth rate of the internet. March 2001. http://www.firstmonday.dk/issues/issue3_10/coffman/.

[15] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of the ACM Symposium on Communications Architectures and Protocols*, pages 1–12, 1989.

[16] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. of IEEE INFOCOM*, pages 636–646, April 1994.

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

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

[19] J. Kaur and H. Vin. Core-stateless guaranteed rate scheduling algorithms. In *Proc. of IEEE INFOCOM*, pages 1484–1492, April 2001.

[20] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 294–303, Dec. 1999.

[21] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow-control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

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

[23] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. Manuscript, Sept. 2002.

[24] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proc. of the 34th Annual ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[25] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. Manuscript, Nov. 2002.

[26] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas. Manuscript, Nov. 2002.

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

[28] I. Stoica and H. Zhang. Providing guaranteed services without per flow management. In *Proc. ACM Sigcomm*, pages 81–94, Sept. 1999.

[29] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.