Fair Integrated Scheduling of Soft Real-time Tardiness Classes on Multiprocessors *

UmaMaheswari C. Devi and James H. Anderson
Department of Computer Science, The University of North Carolina, Chapel Hill, NC

Abstract

Prior work on Pfair scheduling has resulted in three optimal multiprocessor scheduling algorithms, and one algorithm, EPDF, that is less expensive but not optimal. EPDF is still of interest in soft real-time systems, however, due to its ability to guarantee bounded tardiness. In particular, it has been shown that a tardiness bound of t quanta is possible under EPDF if all task weights (*i.e.*, shares or utilizations) are restricted to a value specified as a function of t. In an actual system, however, different tasks may be subject to different tardiness bounds. If such a system is scheduled under EPDF, then the tardiness of a task with a higher bound may cause the tardiness bound of a task with a lower bound to be violated; that is, *temporal isolation* among the various tardiness classes may not be guaranteed. In this paper, we propose an algortihm based on EPDF for scheduling task classes with different tardiness bounds on a multiprocessor. Our algorithm provides temporal isolation among classes, allows the available processing capacity to be fully utilized, and does not require that previously established per-task weight restrictions be made more stringent.

1 Introduction

Pfair scheduling, originally introduced by Baruah *et al.* [4], is the only known way of optimally scheduling recurrent real-time tasks on multiprocessors. Under Pfair scheduling, each task must execute at an approximately uniform rate, while respecting a fixed-size allocation quantum. A task's execution rate is defined by its *weight* (or *utilization*). Uniform rates are ensured by subdividing each task T into quantum-length *subtasks* that are subject to intermediate deadlines. To avoid deadline misses, ties among subtasks with the same deadline must be broken carefully. In fact, tie-breaking rules are of crucial importance when devising optimal Pfair scheduling algorithms.

As discussed by Srinivasan and Anderson [9], overheads associated with tie-breaking rules may be unnecessary or unacceptable for many soft real-time systems. A soft real-time task differs from a hard real-time task in that its deadlines may sometimes be missed. If a job (*i.e.*, task instance) or a subtask with a deadline at time d completes executing at time t, then it is said to have a *tardiness* of $\max(0, t-d)$. As discussed in [7], the results produced by a soft real-time job are of decreasing usefulness after its deadline. Thus, an implicit bound exists on the tardiness that such a job can tolerate.

Systems with quality-of-service requirements, such as multimedia applications, are examples where bounded deadline misses may be tolerable. Here, fair resource allocation is necessary to provide service guarantees, but occasional deadline misses often result in tolerable performance degradation. Hence, an extreme notion of fairness that precludes all deadline misses is usually not warranted.

In dynamic systems that permit tasks to join or leave, the overhead introduced by tie-breaking rules may be unacceptable. In such a system, spare processing capacity may become available that can be reallocated by changing task weights on the fly. It is possible to reweight each task so that its next subtask deadline is preserved [9]. If no tie-breaking information is maintained, then such an approach entails very little overhead. However, weight changes can cause tie-breaking information to change. Thus, if tie-breaking rules are used, then reweighting may necessitate an $\Omega(N \log N)$ cost for N tasks, due to the need to re-sort the scheduler's priority queue. This cost may be prohibitive if load changes are frequent.

The observations above motivated Srinivasan and Anderson to consider the viability of scheduling soft real-time applications using the simpler *earliest-pseudo-deadline-first* (EPDF) Pfair algorithm, which uses no tie-breaking rules. They succeeded in showing that EPDF can guarantee a tardiness of k quanta for every subtask of a feasible task system, if each task's weight is at most $\frac{k}{k+1}$ [9]. In recent work [6], we showed that this condition can be improved to $\frac{k+1}{k+2}$. With either condition, the greater the tardiness allowed, the less stringent the weight restriction.

Contributions. In the work summarized above, *all* tasks are assumed to have equal tolerance to tardiness. How-

*Work supported by NSF grants CCR 9988327, ITR 0082866, CCR 0204312, and CCR 0309825.

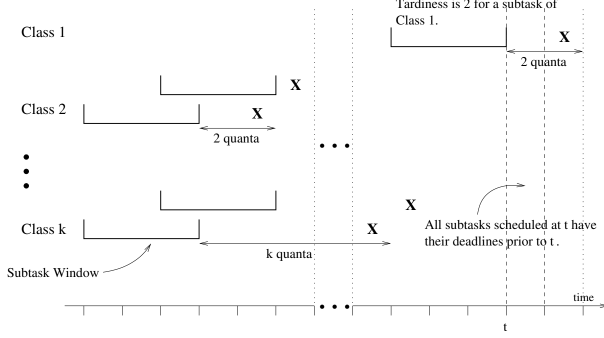


Figure 1. Under Pfair scheduling, each of a task's subtasks has an associated *window* in which it *should* be scheduled; the end of a subtask's window is its deadline. In this figure, a schedule for different classes of soft-real-time tasks on M processors under EPDF is depicted. Tasks in Class i are allowed to miss their deadlines by up to i quanta. For clarity, only a few subtask windows have been shown; for each subtask shown, an 'X' denotes where it is scheduled. At time t, at least M subtasks of Classes 2 and higher with deadlines prior to t have not yet been scheduled. As a result, a subtask of Class 1 with a deadline at t cannot be scheduled at t, and hence, misses its deadline by at least two quanta, *i.e.*, its miss threshold is exceeded.

ever, as discussed in [7], the usefulness of results produced by different soft real-time applications may decrease with tardiness at different rates; thus, different applications can be expected to have different tardiness bounds. When independently-developed real-time applications are multiplexed onto a common platform, it is essential that the temporal correctness of each application is immune from the properties or misbehavior of other applications, *i.e.*, the applications should be *temporally isolated*. If applications with different tardiness bounds are multiplexed, then it should be ensured that the increased demand due to deadline misses in an application with a higher bound does not cause tardiness bounds to be exceeded in an application with a lower bound.

The tardiness bound that can be guaranteed to a task system under EPDF depends on the largest task weight. Hence, if tasks with varying tardiness bounds and weights are present in a system and are scheduled using EPDF, then it may not be possible to guarantee every task its bound. As illustrated in Fig. 1, breaking deadline ties in favor of tasks with more stringent tardiness bounds may not be helpful. Another approach would be to increase the deadline of each subtask by its tardiness bound, and schedule using the modified deadlines. However, it can be shown that, for a reason similar to that illustrated in Fig. 1, this approach may not be successful either. An obvious next solution would be to partition the tasks into classes by their tardiness bounds and schedule each class independently on disjoint sets of processors. Unfortunately, if the total utilization of a class is not integral, then this approach will lead to wasted processing capacity. For example, consider a task system comprised of two tardiness classes with utilizations $M_1+\delta$ and $M_2+1-\delta$, respectively, where $0 < \delta < 1$, $M_1 + M_2 + 1 = M$, and M is integral. Under partitioning, $M_1 + M_2 + 2 = M + 1$ processors will be required to schedule the two classes. Thus, processing capacity equivalent to a full processor would be wasted. In general, with q tardiness classes, $q-1$ additional processors may be required.

In this paper, we propose a new algorithm, based on EPDF, for supporting classes with different tardiness requirements. Our algorithm provides temporal isolation among classes, allows all available processing capacity to be fully utilized, and does not require that previously established per-task weight restrictions be made more stringent. Our algorithm is described in Sec. 3 after first giving needed definitions in Sec. 2. An experimental evaluation of it is presented in Sec. 4.

2 Pfair Scheduling

In this section, we summarize relevant Pfair scheduling concepts and some prior results from [1, 2, 3, 4, 8, 9]. To begin with, we limit attention to periodic tasks that begin execution at time 0. Such a task T has an integer *period* $T.p$, an integer *execution cost* $T.e$, and a *weight* $wt(T) = T.e/T.p$, where $0 < wt(T) < 1$. A task is *light* if its weight is less than $1/2$, and *heavy* otherwise.

Pfair algorithms allocate processor time in discrete quanta; the time interval $[t, t+1)$ 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. The sequence of allocation decisions over time defines a *schedule* S. Formally, $S : \tau \times \mathcal{N} \mapsto \{0, 1\}$, where τ is a task set. $S(T, t) = 1$ iff T is scheduled in slot t. On M processors, $\sum_{T\in\tau} S(T, t) \le M$ 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. Deviation from the fluid schedule is formally captured by the concept of *lag*. Formally, the *lag of task T* at time t is defined by $lag(T, t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u)$.[1] A schedule is defined to be *Pfair* iff

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

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

The lag bounds above have the effect of breaking each task T into an infinite sequence of quantum-length *subtasks*, $T_1, T_2, \ldots$. Each subtask has a *pseudo-release* $r(T_i)$ and a

[1] For conciseness, we leave the schedule implicit and use $lag(T, t)$ instead of $lag(T, t, S)$.

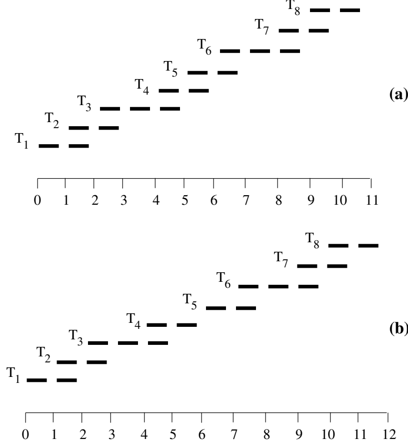


Figure 2. **(a)** Windows of the first job of a periodic task T with weight $8/11$. This job consists of subtasks $T_1, \ldots, T_8$, each of which must be scheduled within its window, or else a lag-bound violation will result. (This pattern repeats for every job.) **(b)** The PF-windows of an IS task. Subtask T_5 becomes eligible one time unit late.

pseudo-deadline $d(T_i)$, where

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \;\; \wedge \;\; d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil .$$

(For brevity, we often omit the prefix "pseudo-.") To satisfy (1), T_i must be scheduled in the interval $w(T_i) = [r(T_i), d(T_i))$, termed its *window*. For example, in Fig. 2(a), $r(T_1) = 0$ and $d(T_1) = 2$. Therefore, T_1 must be scheduled at either time 0 or time 1.

Soft real-time scheduling. As mentioned earlier, the tardiness of a subtask T_i is defined as $tardiness(T_i) = \max(0, t - d(T_i))$, where t is the time that T_i completes execution. The *tardiness of a task system* is then defined as the maximum tardiness among all of its subtasks in any schedule [9].

For reasons discussed in the introduction, we consider scheduling soft tasks using EPDF [9]. EPDF prioritizes subtasks by their deadlines, and resolves any ties arbitrarily. EPDF is not optimal on more than two processors [3], but as discussed earlier, it can ensure a tardiness of at most k (≥ 1) quanta for each subtask, provided certain per-task weight restrictions hold.

Task models. In this paper, we consider the *intra-sporadic* (IS) and the *generalized-intra-sporadic* (GIS) task models [2, 8], which provide a general notion of recurrent execution that subsumes that found in the well-studied periodic and sporadic task models. 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. 2(b). That is, an IS task is obtained by allowing its windows to be shifted right from where they would appear if the task were periodic. A *generalized* intra-sporadic (GIS) task differs from an IS task in that the task may omit some of its subtasks [8].

Each subtask T_i has an additional parameter $e(T_i)$ that specifies the first slot in which it is eligible to be scheduled. We require $e(T_i) \leq r(T_i)$ and $e(T_i) \leq e(T_{i+1})$ for all $i \geq 1$. If $e(T_i) < r(T_i)$ holds, then T_i is said to be *early released*. 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. A schedule for an IS system is *valid* iff each subtask is scheduled in its IS-window. (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.)

As shown in [2], a schedule satisfying (1) on M processors exists for an IS or a GIS task system τ iff $\sum_{T \in \tau} wt(T) \leq M$.

3 Integrating Tardiness Classes

In this section, we present Algorithm I-EPDF, which schedules a soft real-time task system τ comprised of tasks with different tardiness bounds. We let tasks that can be guaranteed a tardiness of c quanta comprise Class c. Thus, if every task can be guaranteed a tardiness of at most q (≥ 1), then there are at most q classes. Any class, except Class q may be empty. If M denotes the total utilization of τ, then I-EPDF schedules τ on at most $\lceil M \rceil$ processors. Without loss of generality, we assume that M is integral. If necessary, this property can be ensured by adding a dummy task of weight $\lceil M \rceil - M$ to τ.

The algorithm consists of three phases: a classification phase, a distribution phase, and a scheduling phase. In the classification phase, the tardiness class of each task is identified, based on its weight. As already mentioned, Srinivasan and Anderson established a per-task weight restriction of $\frac{k}{k+1}$ for ensuring a tardiness of k quanta under EPDF [9], which we later improved to $\frac{k+1}{k+2}$ [6]. In this paper, we assume that task weights are restricted for each class using the $\frac{k}{k+1}$ condition. Our goal in this paper is only to illustrate the idea of integrated scheduling. Our approach is still correct when the $\frac{k+1}{k+2}$ condition is used.

Classification phase. We include in Class c all tasks with weights in the range $(\frac{c-1}{c}, \frac{c}{c+1}]$, *i.e.*, tasks that can be ensured a tardiness of c quanta under EPDF. Note that this has the advantage that a task T with $wt(T) \leq \frac{c}{c+1}$ that can tolerate a tardiness of $d > c$ can be assigned to Class c and guaranteed a lower tardiness bound, without impacting the tardiness of other tasks.

We denote the set of all tasks in Class c by τ^c and its total utilization by M^c. Thus,

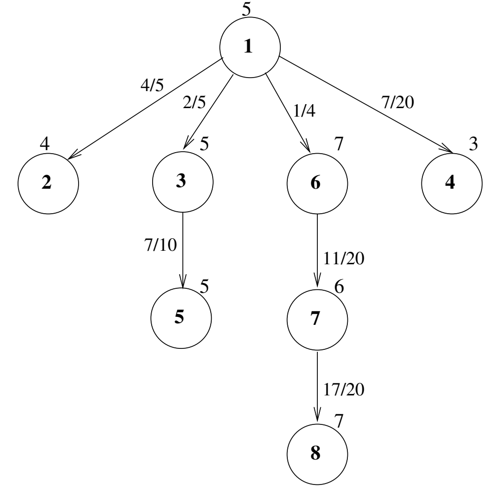
$$\tau = \bigcup_{c=1}^{q} \tau^c, \;\; M^c = \sum_{T \in \tau^c} wt(T), \text{ and } M = \sum_{c=1}^{q} M^c.$$

Class (i)	τ^i	M^i	w^i	Sup_i	λ^i	$\hat{M}^i$	P_i
1	τ^1	$3\frac{1}{5}$	0	0	$\{D^2, D^3, D^4, D^6\}$	5	5
2	τ^2	$4\frac{4}{5}$	$\frac{4}{5}$	1	$\emptyset$	$4\frac{4}{5}$	4
3	τ^3	$4\frac{7}{10}$	$\frac{2}{5}$	1	$\{D^5\}$	$5\frac{7}{10}$	5
4	τ^4	$3\frac{7}{20}$	$\frac{7}{20}$	1	$\emptyset$	$3\frac{7}{20}$	3
5	τ^5	$5\frac{7}{10}$	$\frac{7}{10}$	3	$\emptyset$	$5\frac{7}{10}$	5
6	τ^6	$6\frac{7}{10}$	$\frac{1}{4}$	1	$\{D^7\}$	$7\frac{1}{4}$	7
7	τ^7	$5\frac{7}{10}$	$\frac{11}{20}$	6	$\{D^8\}$	$5\frac{11}{20}$	6
8	τ^8	$7\frac{17}{20}$	$\frac{17}{20}$	7	$\emptyset$	$7\frac{17}{20}$	7

(a)

D^i	w^i
D^1	0
D^2	$\frac{4}{5}$
D^3	$\frac{2}{5}$
D^4	$\frac{7}{20}$
D^5	$\frac{7}{10}$
D^6	$\frac{1}{4}$
D^7	$\frac{11}{20}$
D^8	$\frac{17}{20}$

(b)



(c)

Figure 3. **(a)** Distribution of processors for a system with eight tardiness classes. Column headings refer to various terms mentioned in the text. (We explain in Sec. 3.2 how some of these values are calculated.) **(b)** Weights of donor tasks. **(c)** Tree representation of the task system in (a). Labels within nodes indicate class indices. The integer adjacent to a node denotes the number of processors assigned to the class that the node represents. Edge (a, b) defines the supplier/borrower relation beween Classes a and b; Class a supplies a processing capacity of $w(a, b)$ to Class b, where $w(a, b)$ is the weight of (a, b).

Distribution phase. The goal of this phase is to distribute the M processors among the classes and to define how processors are shared. The processors are divided into q groups of sizes $P_1, \ldots, P_q$, with the ith group assigned to Class i. Because the number of processors assigned to Class i is integral, whereas its total utilization M^i may not be, each class is allowed to borrow processing capacity from at most one lower-indexed class. To ensure correctness for each class, this borrowing is subject to a number of rules given below. Later, in Sec. 3.2, we present an algorithm for defining an assignment that satisfies these rules. Before stating these rules, we introduce some relevant notation. The use of this notation is illustrated in Fig. 3.

Notation. w^i denotes the amount of processing capacity that Class i borrows from some lower-indexed class. Sup_i denotes the lower-indexed class that supplies processing capacity to Class i; if $w^i = 0$, then $Sup_i = 0$. f^i denotes the fractional part of the utilization of τ^i, *i.e.*,

$$f^i = M^i - \lfloor M^i \rfloor. \tag{2}$$

To enable the different classes to share processors at runtime, a donor task D^j $(1 \le j \le q)$ of weight $w^j > 0$ may be created; D^j is added to Class i, where $i = Sup_j$. (The manner in which D^j is used to share processors is explained in Sec. 3.1.) The set of all donor tasks added to Class i is denoted λ^i. $\hat{\tau}^i$ extends τ^i by including these tasks, *i.e.*,

$$\hat{\tau}^i = \tau^i \cup \lambda^i. \tag{3}$$

Correspondingly, we define

$$\hat{M}^i = M^i + \sum_{T \in \lambda^i} wt(T). \tag{4}$$

Processor sharing rules.[2] The sharing of processors among classes is governed by the following rules.

(R1) The processing capacity that Class i borrows is at most the fractional part of its utilization, *i.e.*,

$$0 \le w^i \le f^i. \tag{5}$$

(R2) Class i borrows processing capacity from at most one class with a lower tardiness bound (*i.e.*, a lower-indexed class), and lends to zero or more classes. In other words, the following hold.

$$(\forall i : i \ge 1 :: Sup_i < i) \tag{6}$$
$$(\forall i : i \ge 1 :: \{j \mid D^i \in \lambda^j\} = \{Sup_i\}) \tag{7}$$
$$(\forall i :: |\{j : Sup_j = i\}| \ge 0) \tag{8}$$

(R3) If $f^i \le 2/3$ or $w^i = f^i$ holds, where $i \ge 3$, then Class i does not lend any processing capacity to other classes. If $0 < f^i \le 1/2$, then Class i borrows a capacity of f^i from Class 1; if f^i ranges between 1/2 and 2/3, then it borrows f^i from Class 2. If $f^i = 0$, then Class i does not borrow.

$$(\forall i : i \ge 3 \ \wedge \ 0 < f^i \le 1/2 :: w^i = f^i \ \wedge \ Sup_i = 1) \tag{9}$$
$$(\forall i : i \ge 3 \ \wedge \ 1/2 < f^i \le 2/3 :: w^i = f^i \ \wedge \ Sup_i = 2) \tag{10}$$
$$(\forall i : i \ge 3 \ \wedge \ f^i = 0 :: w^i = 0 \ \wedge \ Sup_i = 0) \tag{11}$$
$$(\forall i : i \ge 3 :: w^i = f^i \iff \lambda^i = \emptyset) \tag{12}$$

(R4) The processing capacity that Class 3 or higher borrows is less than what it lends to any single class, *i.e.*,

$$(\forall i : i \ge 3 :: (\forall j : Sup_j = i \Rightarrow w^i < w^j)). \tag{13}$$

[2] Some of these rules are somewhat technical in nature. They are included to address certain cases that arise in showing that I-EPDF is correct.

```
ALGORITHM I-EPDF(τ)
        ℰ_1, ..., ℰ_q: integer;
        P_1, ..., P_q: integer;
        D^1, ..., D^q: GIS tasks initially ∅;
        τ^1, ..., τ^q: GIS task sets initially ∅;
        τ̂^1, ..., τ̂^q: GIS task sets initially ∅;
        λ^1, ..., λ^q: GIS task sets initially ∅;
        tight_1, ..., tight_q: boolean initially TRUE

1   Classification Phase
2   Distribution Phase
    Scheduling Phase
3   t := 0;
4   while TRUE do
5     for i := q downto 1 do
6       if τ̂^i ≠ ∅ then
7         for each D^c in λ^i do
8           if ℰ_c ≤ P_c then
9             s := index of next eligible subtask of D^c;
10            if r(D^c_s) ≤ t ∧ d(D^c_s) > t + 1 then
11              if r(D^c_s) < t) then s := 1 fi;
12              r(D^c_s), e(D^c_s) := t + 1, t + 1
              fi
            fi
          od;
13        ℰ_i := # of eligible tasks in τ̂^i excluding those
                 early-released (it suffices to determine
                 if P_i + 1 are eligible)
        fi
      od;
14    for i := 1 to q do /*The lowest-indexed class is always tight*/
15      if tight_i then
16        maxSchedulable := P_i
        else
17        maxSchedulable := P_i + 1
        fi;
18      Schedule at most maxSchedulable tasks of
        τ̂^i using EPDF (for Class 3 and higher, break ties
        involving the heavy donor task, if any,
        in favor of the donor task);
19      for each D^c in λ^i do
20        if D^c is scheduled then
21          tight_c := FALSE
          else
22          tight_c := TRUE
          fi
        od
      od;
23    t := t + 1
    od
```

Figure 4. Algorithm I-EPDF—detailed pseudo-code for the scheduling phase.

(R5) The number of processors assigned to the various classes must satisfy the following.

$$(\forall i : P_i = M^i - w^i + \sum_{\{j:Sup_j=i\}} w^j) \quad (14)$$

$$P_1 = \lceil M^1 + w^2 + \textstyle\sum_{\{j:((j\geq 3)\ \wedge\ (f^j \leq 1/2))\}} w^j \rceil \quad (15)$$

$$P_2 = \lfloor M^2 + \textstyle\sum_{\{j:((j\geq 3)\ \wedge\ (1/2<f^j \leq 2/3))\}} w^j \rfloor \quad (16)$$

$$(\forall i : i \geq 3 :: P_i = \lceil M^i \rceil \vee\ P_i = \lfloor M^i \rfloor) \quad (17)$$

$$\sum_{i=1}^{q} P_i = M \quad (18)$$

The supplier/borrower relationship among classes can be represented as a forest of weighted trees in which nodes represent classes. An edge of weight w between nodes i and j, where $i < j$, implies that $w^j = w$ and $Sup_j = i$. As an example, Fig. 3 shows an assignment of processors to classes and a supplier/borrower relation among classes that conforms to the rules above for a task system comprised of eight tardiness classes.

3.1 Scheduling Phase of I-EPDF

Assuming that processors are assigned to classes per the rules above, we now explain the scheduling phase. As mentioned earlier, Sec. 3.2 presents an algorithm for creating such an assignment. In the scheduling phase, a separate instantiation of EPDF is used to schedule each $\hat{\tau}^i$. The pseudo-code for this phase is shown in lines 3–23 in Fig. 4. $\hat{\tau}^i$ includes the donor tasks in λ^i (refer (3)), which compete with tasks in τ^i at every time instant. If at time t, a donor task D^j in λ^i is scheduled, then one of the processors of Class i is handed down to Class j. Thus, Class j has $P_j + 1$ processors for scheduling the tasks in $\hat{\tau}^j$ at time t, zero or more of which may be handed down to higher-indexed classes, recursively.

In the first part of the scheduling phase given by the **for** loop of lines 5–13, the number of eligible subtasks of $\hat{\tau}^c$ at time t is identified in the iteration in which $i = c$. Because the **for** loop considers the classes in decreasing index order, the number of eligible tasks in classes with higher indices than c are known at this time. By (6) and (7), donor tasks in λ^c are of higher index than c. Therefore, if $\hat{\tau}^c$ includes donor task D^k and the number of eligible tasks in $\hat{\tau}^k$ is at most P_k, then the release time of the next subtask D^k_s of D^k is postponed to $t + 1$, if its deadline is greater than $t + 1$. We do this because Class k is not able to use an extra processor that it would be given, and hence, by postponing the release time of the next subtask of its donor task under the conditions specified, Class k may be provided with an extra processor sooner in the future than may otherwise be possible. Another related rule in line 11 is that if the release time of D^k_s before the postponement was earlier than t, then D^k_s is replaced by D^k_1 with $r(D^k_1)$ set to $t + 1$.

The second part of the scheduling phase, given by the **for** loop in lines 14–23, determines the maximum number of tasks of $\hat{\tau}^c$ that can be scheduled ($maxSchedulable$) at t, and schedules those with the highest priority. For all but the lowest-indexed class, $maxSchedulable$ is either P_c or $P_c + 1$, based on whether D^c is scheduled. If P_c processors are available for scheduling tasks in $\hat{\tau}^c$, then t is said to be a *tight* slot for $\hat{\tau}^c$; otherwise, t is a *non-tight* slot for $\hat{\tau}^c$. An example is given in Fig. 5, in which Classes 1 and 6 from

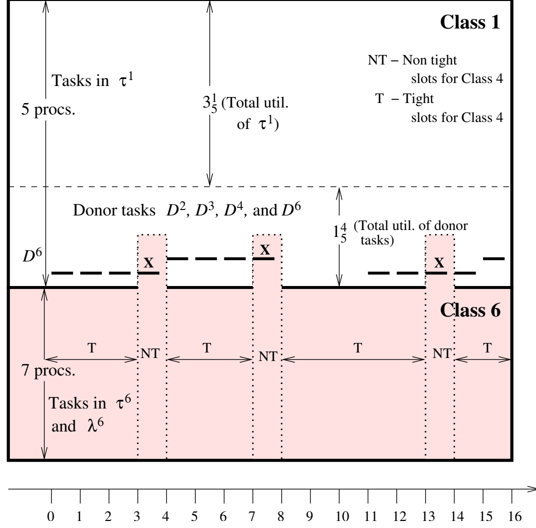


Figure 5. Classes 1 and 6 of the example task system in Fig. 3. Class 1 is assigned five processors and supplies processing capacity to Classes 2, 3, 4, and 6. Class 6 is assigned seven processors and borrows a processing capacity of $1/4$ from Class 1. An additional processor is handed down to Class 6 from Class 1 when donor task D^6 is scheduled in Class 1. The partial schedule depicted shows the first three subtasks of D^6, which are scheduled in the slots marked by an 'X'. The release of the third subtask is postponed from time 8 to time 11. Thus, slots 4, 8, and 14 are non-tight for Class 6 and the rest are tight.

Fig. 3 are considered.

Selecting the highest priority tasks to schedule dominates the per-slot time complexity of this phase, and hence, it is the same as that of EPDF, *i.e.*, $O(M \log N)$. A correctness proof is presented in [5].

3.2 Distribution Phase of I-EPDF

In this subsection, we describe an algorithm for distributing the processors and defining supplier/borrower relations that conform to (R1)–(R5). Fig. 6 presents the detailed pseudo-code. In describing this code, we refer to Class j as an i*-borrower* if Class j borrows processing capacity from Class i. The computation here proceeds in three steps.

The **for** loop in lines 1–7 comprises the first step, which ensures that (9), (10), and (11) of (R3) hold, and together with the third step (see below), that (12) and (17) are satisfied. In this step, Class i, where $i \geq 3$, is set to borrow its entire fractional utilization of f^i from Class 1, if $f^i \leq 1/2$, or from Class 2, if $1/2 < f^i \leq 2/3$. This is done by setting $w^i = M^i - \lfloor M^i \rfloor = f^i$ (refer (2)) in line 3, followed by the addition of a donor task D^i of appropriate weight to Class 1 or Class 2. Every class that is made a 1- or a 2-borrower at the end of Step 1, is considered *finished*, and does not participate in future distribution steps. (More specifically, Class i marked *finished* by setting the boolean variable $done_i$ to TRUE cannot have new donor tasks added to it.) Such a class is assigned $\lfloor M^i \rfloor$ processors, which are not shared with other classes. For example, consider Fig. 3. The total utilization of τ^4 is $3\frac{7}{20}$, with $f^4 = \frac{7}{20} < \frac{1}{2}$. Therefore, at the end of the step just described, a donor task D^4 of weight $w^4 = \frac{7}{20}$ is added to Class 1, three processors are assigned to Class 4, and it is marked finished.

Lines 8–12 comprise the next step, which ensures (16). In this step, Class 2 is made a 1-borrower by letting it borrow a processing capacity of $w^2 = (M^2 + \sum_{T \in \lambda^2} wt(T)) - \lfloor (M^2 + \sum_{T \in \lambda^2} wt(T)) \rfloor$ from Class 1 (where by (3), $M^2 + \sum_{T \in \lambda^2} wt(T) = \hat{M}^2$), and is assigned $\lfloor \hat{M}^2 \rfloor$ processors; it is then marked finished. In the example in Fig. 3, the fractional part of the utilization of no class is between $1/2$ and $2/3$; therefore, no donor tasks are added to Class 2 in the previous step. Thus, $w^2 = \frac{4}{5}$, a donor task D^2 of weight $\frac{4}{5}$ is added to Class 1, Class 2 is assigned four processors, and is marked finished.

The **while** loop in lines 14–40 constitutes the third step in the distribution phase. This step is responsible for ensuring (12), (13), (15), and (17). In this step, every class that is not yet finished is considered in increasing index order in an iteration of the **while** loop. Classes that are already finished are skipped via line 15. Thus, the unfinished class with the lowest index is given by the value of i at the end of this line. The goal of the remainder of the iteration is to determine at most two higher-indexed classes with which Class i can share its spare capacity (given by $\lceil \hat{M}^i - w^i \rceil - (\hat{M}^i - w^i)$). (To ensure the tardiness bound of the borrowing class, it is necessary to ensure that it does not borrow from a class with a larger bound.) Class i is also marked finished at the end of the ith iteration. Thus, at the beginning of iteration i, every class with a lower index than i is already finished. Note that for Class 3 and higher, $\hat{M}^i = M^i$ holds at line 17. This is because these classes are not augmented with donor tasks prior to this point. Line 19 identifies Class l with the lowest index greater than i that is not finished that can be made an i-borrower. To ensure (R1), if $f^l \leq avail$ holds, then Class l is made to borrow a processing capacity of f^l from Class i and is marked finished in the **if** block in lines 21–27. Line 27 identifies and sets l to the next higher-indexed class that is not yet finished.

Regardless of whether the test in line 21 succeeded, $f^l > avail$ holds at line 28. This is clearly the case if the test in line 21 failed. On the other hand, if this test succeeded, then because $f^l > 2/3$ holds for every class of index three or higher that is not finished by Step 2, $w^l > 2/3$ holds at line 25. Because $avail$ as computed in line 17, is less than one, $avail < 1/3$ holds at the end of line 25. Hence, for the same reason that $f^l > 2/3$ holds for every class of index exceeding three that is not finished by Step 2, $f^l > avail$ holds at line 28. Because the amount of processing capacity that Class l borrows is set to $\min(avail, f^l)$, Class i can

Distribution Phase of ALGORITHM I-EPDF(τ)

$Sup_1, \ldots, Sup_q$: **integer initially** 0;
$done_1, \ldots, done_{q+1}$: **boolean initially** FALSE

1 **for** $i := 3$ **to** q **do**
2   **if** $M^i - \lfloor M^i \rfloor \leq 2/3$ **then**
3     $w^i := M^i - \lfloor M^i \rfloor$;
4   **if** $w^i > 0$ **then**
5   **if** $w^i \leq 1/2$ **then** $Sup_i := 1$ **else** $Sup_i := 2$ **fi**;
6   ADDDONORTASK(i, w^i, Sup_i)
      fi;
7     $P_i,\ done_i := \lfloor M^i \rfloor,$ TRUE
    fi
od;
8 $w^2 := \hat{M}^2 - \lfloor \hat{M}^2 \rfloor$;
9 **if** $w^2 > 0$ **then** ADDDONORTASK$(2, w^2, 1)$ **fi**;
10 $P_2,\ done_2 := \lfloor \hat{M}^2 \rfloor,$ TRUE;
11 **if** $\lceil \hat{M}^1 \rceil = \hat{M}^1$ **then**
12   $P_1,\ done_1 := \hat{M}^1,$ TRUE
fi;
13 $i := 1$;
14 **while** $i \leq q$ **do**
15   **while** $done_i$ **do** $i := i + 1$ **od**;
16   **if** $i \leq q$ **then**
17     $avail := \lceil \hat{M}^i - w^i \rceil - (\hat{M}^i - w^i)$;
18     $l := i + 1$;
19   **while** $done_l$ **do** $l := l + 1$ **od**;
20   **if** $l \leq q$ **then**
21   **if** $avail > 0\ \wedge\ (M^l - \lfloor M^l \rfloor) \leq avail$ **then**
22         $w^l := M^l - \lfloor M^l \rfloor$;
23   ADDDONORTASK(l, w^l, i);
24         $P_l,\ done_l := \lfloor M^l \rfloor,$ TRUE;
25         $avail := avail - w^l$;
26         $l := l + 1$;
27   **while** $done_l$ **do** $l := l + 1$ **od**
      fi;
28   **if** $avail > 0$ /* $\wedge\ l \leq q$ */ **then**
29         $w^l := avail$;
30   ADDDONORTASK(l, w^l, i);
31         $d,\ j := l,\ i$;
32   **while** $w^d < w^j$ **do**
33           $\hat{\tau}^j,\ \lambda^j := \hat{\tau}^j - \{D^d\},\ \lambda^j - \{D^d\}$;
34           $w^j,\ \hat{M}^j := w^j - w^d,\ \hat{M}^j - w^d$;
35           $\hat{\tau}^{Sup^j} := \hat{\tau}^{Sup^j} \cup \{D^d\}$;
36           $\lambda^{Sup^j} := \lambda^{Sup^j} \cup \{D^d\}$;
37   **if** $w^j < w^d$ **then** $d := j$ **fi**;
38           $j := Sup_j$
        od
      fi;
39       $P_i,\ done_i := \lfloor \hat{M}^i \rfloor,$ TRUE
    fi
40     $i := l$
  fi
od

PROCEDURE ADDDONORTASK(i, w, sup)
41 $D^i :=$ donor task of weight w;
42 $Sup_i := sup$;
43 $\hat{\tau}^{sup},\ \lambda^{sup} := \hat{\tau}^{sup} \cup \{D^i\},\ \lambda^{sup} \cup \{D^i\}$;
44 $\hat{M}^{sup} := \hat{M}^{sup} + w^i$;
45 **return**

Figure 6. Algorithm I-EPDF—detailed pseudo-code for the distribution phase.

have at most two donor tasks added to it in this iteration. l is the unfinished class with the lowest index at line 40. Therefore, i is updated to l so that Class l is considered for the addition of donor tasks in the next iteration.

Using our example (Fig. 3), $avail$ at line 17 when $i = 1$ holds is $\lceil \hat{M}^1 \rceil - \hat{M}^1 = 5 - 4\frac{7}{20} = \frac{13}{20}$. (Because D^4 of weight $w^4 = \frac{7}{20}$ and D^2 of weight $w^2 = \frac{4}{5}$ were added to Class 1 in Steps 1 and 2, respectively, by (4), $\hat{M}^1 = M^1 + w^4 + w^2 = 3\frac{1}{5} + \frac{7}{20} + \frac{4}{5} = 4\frac{7}{20}$, and hence, $\lceil \hat{M}^1 \rceil = 5$.) In this iteration of the **while** loop in lines 14–40, the first unfinished class with a higher index than one, which is Class 3, is made a 1-borrower (lines 29–30). Thus, $l = 3$ in this case. Hence, at the end of the first iteration, w^3 is set to $\frac{13}{20}$ (line 29) and donor task D^3 of weight w^3 is added to Class 1 (line 30). Class 1 is then marked finished (line 39). The unfulfilled utilization of Class 3 is now $4\frac{7}{10} - \frac{13}{20} = 4\frac{1}{20}$. Therefore, Class 3 is assigned five processors and has a spare capacity of $\frac{19}{20}$. Because Class 3 is the next unfinished class, classes with which its spare capacity is shared are identified in the next iteration.

One final adjustment is performed in lines 32–38. If the weight of the donor task w^l that is added to Class i is less than w^i, *i.e.*, the processing capacity that Class i borrows in turn from its supplier $j = Sup_i$, then Class j is made Class l's supplier, too. This is done by moving D^l to Class j from Class i. w^i is appropriately reduced so that the total capacity that Class j supplies remains the same. As a result of this adjustment, Class j will now have two donor tasks D^i and D^l in place of D^i (it is possible that Class j has some other donor tasks, whose weights are not altered), and it is possible for one of them to be lighter than D^j. If this is the case, then the **while** loop in lines 32–38 moves the lighter of the two donor tasks, D^i and D^l, up the supplier chain, to ensure (R4). Fig. 3 shows the final distribution and sharing of processors among classes.

It can be shown that the complexity of the above algorithm is $\Theta(q)$. A correctness proof is presented in [5].

4 Experimental Evaluation

In this section, we report results of our empirical evaluation of the additional processing capacity that may be required when classes do not share processors. The evaluation procedure was as follows. 1,000,000 task sets were generated at random, with total utilization M in the range 5..64. The tasks in each task set were divided into q tardiness classes based on their weights. The total number of processors P required to schedule the task set was then computed, assuming that each tardiness class has exclusive access to the processors that it requires. The difference $E = P - M$, which represents the additional processing capacity required, was then determined. The average value of E (expressed as a percentage of M) with respect to total

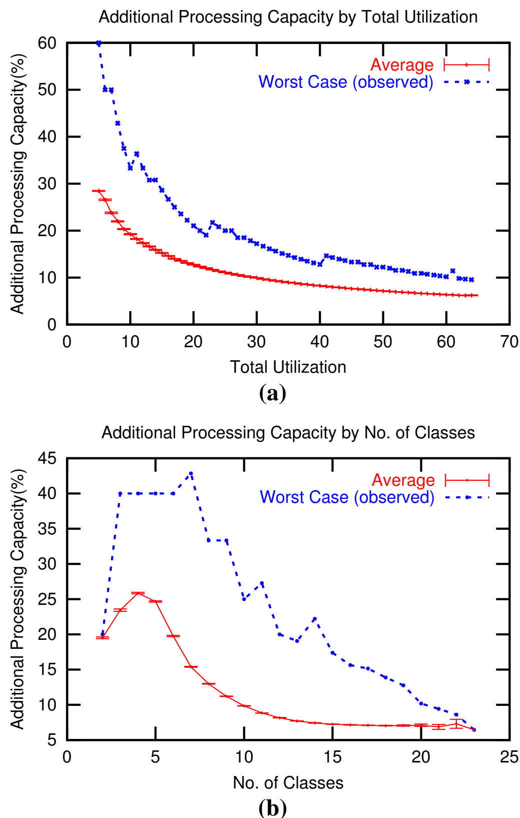


Figure 7. Additional processing capacity (as a percentage of total utilization) with respect to **(a)** total utilization and **(b)** number of classes that may be required if tardiness classes are partitioned rather than using I-EPDF.

utilization (M) and the number of classes (q) is plotted in Fig. 7. 99% confidence intervals are also shown on these plots. The figure also depicts the worst-case observed values of E, for each value of M and q. The graphs show that the average percentage of loss is quite high (over 30%), for small values of M and q, and decreases with increasing M and q. The reason for this is as follows. In Fig. 7(a), the value of q for a given M is the average over all task sets with that value of M, and in Fig. 7(b), the value of M for a given q is the average over all task sets with that value of q. As mentioned in the introduction, E is at most $q - 1$. Also, because a class can span multiple processors, q increases at a lower rate than M. Therefore, E, expressed as a percentage of M, decreases with increasing q and M. Even though E decreases as M and q increase, the loss is still more than 5% for large M and q, which suggests significant waste.

5 Conclusion

We have presented a new algorithm for integrating soft real-time tardiness classes on a multiprocessor. Our algorithm provides temporal isolation among classes, allows available processing capacity to be fully utilized, and does not require that previously established per-task weight restrictions for a given tardiness threshold be lowered. Our experiments indicate that the proposed algorithm allows a substantial amount of processing capacity to be reclaimed.

Our algorithm can be extended to allow hard tasks, in addition to soft tasks. In that case, an optimal algorithm (with tie breaks) is used for scheduling hard tasks, while a separate instantiation of EPDF is used for each soft class.

As discussed in Sec. 1, one motivation for using EPDF is the ability to reweight tasks efficiently in dynamic systems. However, reweighting a task may alter the tardiness bound that can be guaranteed to it, and hence, may require that the task be migrated to a different tardiness class. Redistributing processors to the redefined classes can be done in constant time. It only remains to be proved that the tardiness bounds of individual tasks can still be guaranteed. We are currently working on this problem.

Acknowledgement: We are thankful to Phil Holman for his suggestions on improving the presentation of this paper.

References

[1] 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.

[2] 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.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

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

[5] U. Devi and J. Anderson. Fair integrated scheduling of soft real-time tardiness classes on multiprocessors (full paper). Available at http://www.cs.unc.edu/~anderson/papers.html, Feb 2004.

[6] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. In *Proc. of the 12th International Workshop on Parallel and Distributed Real-time Systems*, April 2004. To Appear.

[7] J.W.S. Liu. *Real-time Systems*. Prentice Hall, 2000.

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

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