An EDF-based Restricted-Migration Scheduling Algorithm for Multiprocessor Soft Real-Time Systems*

James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi

Department of Computer Science

The University of North Carolina at Chapel Hill, Chapel Hill, NC 27599, USA

Abstract

There has been much recent interest in the use of the *earliest-deadline-first* (EDF) algorithm for scheduling soft real-time sporadic task systems on identical multiprocessors. In hard real-time systems, a significant disparity exists between EDF-based schemes and Pfair scheduling: on M processors, the worst-case schedulable utilization for all known EDF variants is approximately $M/2$, whereas it is M for optimal Pfair algorithms. This is unfortunate because EDF-based algorithms entail lower scheduling and task-migration overheads. However, such a disparity in schedulability can be alleviated by easing the requirement that all deadlines be met, which may be sufficient for soft real-time systems. In particular, in recent work, we have shown that if task migrations are not restricted, then EDF (*i.e.*, global EDF) can ensure bounded tardiness for a sporadic task system with no restrictions on total utilization. Unrestricted task migrations in global EDF may be unappealing for some systems, but if migrations are forbidden entirely, then bounded tardiness cannot be guaranteed. In this paper, we address the issue of striking a balance between task migrations and system utilization by proposing an algorithm called EDF-fm, which is based upon EDF and treads a middle path, by restricting, but not eliminating, task migrations. Specifically, under EDF-fm, the ability to migrate is required for at most $M - 1$ tasks, and it is sufficient that every such task migrate between two processors and at job boundaries only. EDF-fm, like global EDF, can ensure bounded tardiness to a sporadic task system as long as the available processing capacity is not exceeded, but, unlike global EDF, may require that per-task utilizations be capped. The required cap is quite liberal, hence, EDF-fm should enable a wide range of soft real-time applications to be scheduled with no constraints on total utilization.

*Work supported by NSF grants CCR 0204312, CNS 0309825, CNS 0408996, and CNS 0615197, and by ARO grant W911NF-06-1-0425. The third author was also supported by an IBM Ph.D. fellowship. A preliminary version of this paper was published in the Proceedings of the 17th Euromicro Conference on Real-Time Systems [3].

1 Introduction

Multiprocessor-based real-time systems are now commonplace. Designs range from single-chip architectures, with a modest number of processors, to large-scale signal-processing systems, such as synthetic-aperture radar systems. In recent years, scheduling techniques for such systems have received considerable attention. In an effort to catalogue these various techniques, Carpenter *et al.* [13] suggested the categorization shown in Table 1, which pertains to scheduling algorithms for *periodic* or *sporadic* task systems. In such systems, each task is invoked repeatedly, and each such invocation is called a *job*. The table classifies scheduling algorithms along two dimensions:

1. **Complexity of the priority mechanism.** Along this dimension, scheduling algorithms are categorized according to whether task priorities are **(i)** static, **(ii)** dynamic but fixed within a job, or **(iii)** fully-dynamic. Common examples of each type include (i) rate-monotonic (RM) [28], (ii) earliest-deadline-first (EDF) [28], and (iii) least-laxity-first (LLF) [32] scheduling.

2. **Degree of migration allowed.** Along this dimension, algorithms are classified as follows: **(i)** no migration (*i.e.*, task partitioning), **(ii)** migration allowed, but only at job boundaries (*i.e.*, dynamic partitioning at the job level), and **(iii)** unrestricted migration (*i.e.*, jobs are also allowed to migrate).

The entries in Table 1 give known upper and lower bounds on *schedulable utilization* for each category, assuming that jobs can be preempted and resumed later. If $U(M)$ is a schedulable utilization for an M-processor scheduling algorithm $\mathcal{A}$, then $\mathcal{A}$ can correctly schedule any set of periodic (or sporadic) tasks with total utilization not exceeding $U(M)$ on M processors. The top left entry in the table means that there exists some algorithm in the unrestricted-migration/static-priority class that can correctly schedule every task set with total utilization at most $\frac{M^2}{3M-2}$, and that there exists some task set with total utilization slightly higher than $\frac{M+1}{2}$ that cannot be correctly scheduled by any algorithm in the same class. The other entries in the table have a similar interpretation.

According to Table 1, scheduling algorithms from only one category can be *optimal*, *i.e.*, can schedule tasks correctly with no utilization loss, namely, algorithms that allow full migration and use fully-dynamic priorities (the top right entry). The known optimal algorithms in this category use the *proportionate fair* or Pfair approach [11] for scheduling. Pfair algorithms break tasks into smaller uniform pieces called "subtasks," which are then scheduled. The subtasks of a task may execute on any processor, *i.e.*, tasks may migrate within jobs. Hence, Pfair scheduling algorithms may suffer from higher scheduling and migration overheads than other schemes. Thus, the other categories in Table 1 are still of interest.

In five of the other categories, the term α represents a cap on individual task utilizations. If such a cap is not exploited, then as shown in [13], on $M \geq 2$ processors, *no* algorithm in these categories can successfully schedule all task systems with total utilization exceeding $(M+1)/2$. Given the scheduling and migration overheads of Pfair

	P_s**: static**	P_d^r**: job-level fixed/task-level dynamic**	P_d^u**: fully dynamic**
M_f**: full migration**	$\frac{M^2}{3M-2} \leq U \leq \frac{M+1}{2}$ [4]	$U = M - \alpha(M-1)$, if $\alpha \leq \frac{1}{2}$ [19] $U = \frac{M+1}{2}$, otherwise [9]	$U = M$ [11, 36]
M_r**: restricted migration**	$U \leq \frac{M+1}{2}$	$U \geq M - \alpha(M-1)$, if $\alpha \leq \frac{1}{2}$ [10] $U = \frac{M+1}{2}$, otherwise [10]	$U \geq M - \alpha(M-1)$, if $\alpha \leq \frac{1}{2}$ $U = \frac{M+1}{2}$, otherwise
M_p**: partitioned**	$U = \frac{M+1}{2}$ [5]	$U = \frac{\beta M+1}{\beta+1}$, where $\beta = \left\lfloor \frac{1}{\alpha} \right\rfloor$ [30]	$U = \frac{\beta M+1}{\beta+1}$, where $\beta = \left\lfloor \frac{1}{\alpha} \right\rfloor$

Table 1: (Table from [13]. Some entries have been updated to reflect later advances.) Known lower and upper bounds on the worst-case schedulable utilization on M processors, $U(M)$ (denoted U in the table), for the different classes of scheduling algorithms. $\alpha = u_{\max}$, the maximum utilization of any task in the task system under consideration. Citations next to the entries indicate the source of each bound.

algorithms, the disparity in schedulability between Pfair algorithms and those in other categories is somewhat disappointing.

Fortunately, as the table suggests, if individual task utilizations can be capped, then it is sometimes possible to significantly relax restrictions on total utilization. For example, in the entries in the middle column, as α approaches 0, $U(M)$ approaches M. This follows from work on EDF scheduling on multiprocessors [8, 10, 31], which shows that an interesting "middle ground" exists between the worst-case schedulable utilizations of EDF-based algorithms (which is $M/2$ approximately) and Pfair algorithms (which is M). In essence, establishing this middle ground involved addressing the following question: *if per-task utilizations are restricted, and if no deadlines can be missed, then what is the largest overall utilization that can be allowed?* This middle ground can be approached in a different way by addressing an alternative question: *if per-task utilizations are restricted, but overall utilization is not, then by how much can deadlines be missed?* Our interest in this question stems from the increasing prevalence of applications such as networking, multimedia, and immersive graphics systems (to name a few) that have only soft real-time requirements.

In related work (which followed the publication of the results herein in preliminary form [3]), we have shown that if tasks are not pinned to processors and may migrate freely, then both preemptive and non-preemptive global EDF can guarantee bounded tardiness while requiring no restrictions on either per-task utilizations or the total system utilization [16]. Tardiness bounds have been derived under preemptive EDF by Valente and Lipari also [39]. However, unrestricted migration may be undesirable in some systems, such as those that have large working sets, and hence, high migration costs. On the other hand, a no-migration algorithm has no scope of improving system utilization even if bounded tardiness is tolerable. This is because, if a task system cannot be partitioned among the available processors without overutilizing some processor, then deadline misses and tardiness for the tasks assigned to that processor will increase with time. In this paper, we address the issue of finding an acceptable middle ground.

Contributions of this paper. Our first contribution is the design of an algorithm called EDF-fm, which treads a middle path between full-migration and no-migration algorithms by restricting the number of tasks that need to migrate, and the derivation of a tardiness bound that can be guaranteed under it. The tardiness bound derived can be computed in $\mathcal{O}(N)$ time, where N is the number of tasks. Though our basic scheme adheres to the conditions of the middle entry of Table 1 (restricted migration, job-level dynamic priorities), the degree of migration that is needed is in fact lower than that suggested by that entry: under EDF-fm, only up to $M-1$ tasks, where M is the number of processors, *ever* migrate, and those that do, do so only between jobs and only between two processors. As noted in [13], migrations between jobs should not be much of a concern for tasks for which little state is carried over from one job to the next.

The maximum tardiness that any task may experience under EDF-fm is dependent on the per-task utilization cap assumed—the lower the cap, the lower the tardiness threshold. Even with a cap as high as 0.5 (*half* of the capacity of one processor), reasonable tardiness bounds can be guaranteed for a significant percentage of task systems. (In contrast, if $\alpha = 0.5$ in the middle entry of Table 1, then approximately 50% of the system's overall capacity may be lost.) Hence, our scheme should enable a wide range of soft real-time applications to be scheduled in practice with *no constraints on total utilization*. In addition, when a job misses its deadline, we do *not* require a commensurate delay in the release of the next job of the same task. As a result, each task's required processor share is maintained in the long term.

As a second contribution, we propose several heuristics for assigning to processors those tasks that do not migrate under EDF-fm, and, through extensive simulations, evaluate the efficacy of these heuristics in lowering the tardiness bound that can be guaranteed. We also present a simulation-based evaluation of the accuracy of the tardiness bound under the heuristic identified to be the best. Finally, we provide a set of iterative formulas, which may potentially require exponential time, for computing a tardiness bound that is less pessimistic than the $\mathcal{O}(N)$-time bound referred to earlier, and evaluate its accuracy through simulations.

Organization. The rest of this paper is organized as follows. Section 2 describes the system model used in this paper. This is followed in Section 3 by a very brief overview of Pfair scheduling that reviews concepts used in the design of EDF-fm. (This section may be skipped during an initial reading and referred to when needed in Section 4.2.2.) In Section 4, Algorithm EDF-fm is described and Section 5 derives a tardiness bound under it. Techniques and heuristics that can be used to reduce tardiness observed in practice, and exponential-time iterative formulas for computing more accurate tardiness bounds, as described above, are presented in Section 6. Then, in Section 7, a simulation-based evaluation of our basic algorithm and proposed heuristics is presented, and the accuracy of the tardiness bound derived is assessed. Finally, after related work is reviewed in Section 8, Section 9 concludes.

2 System Model

A *sporadic task system* [32] τ consisting of $N > M$ independent, sporadic tasks is to be scheduled upon a multiprocessor platform with $M \geq 2$ identical processors. The k^{th} processor is denoted P_k, where $1 \leq k \leq M$. Each task $\tau_i(e_i, p_i)$, where $1 \leq i \leq N$, is characterized by a minimum inter-arrival time, also referred to as its *period*, $p_i > 0$, a *worst-case execution cost* $e_i \leq p_i$, and a *relative deadline* D_i. In this paper we assume that $D_i = p_i$ holds for all i. Every task τ_i may be invoked zero or more times with two consecutive invocations separated by at least p_i time units. Each invocation of τ_i is referred to as a *job* of τ_i and the k^{th} job of τ_i, where $k \geq 1$, is denoted $\tau_{i,k}$. The first job may be invoked or *released* at any time at or after time zero. The release time of job $\tau_{i,k}$ is denoted $r_{i,k}$. A *periodic task system*, in which every two consecutive jobs of every task τ_i are separated by exactly p_i time units, is a special case of a sporadic task system. Every job of τ_i executes for at most e_i time units. The *absolute deadline* (or simply, *deadline*) of $\tau_{i,k}$, denoted $d_{i,k}$ and given by $r_{i,k} + D_i$, is the time at or before which $\tau_{i,k}$ should complete execution. Each task is sequential, and at any time may execute on at most one processor. The *utilization* of τ_i is given by $u_i = e_i/p_i$. τ_i is said to be a *light task* if $0 < u_i \leq 1/2$ holds. The *total utilization* of τ is defined as $U_{sum}(\tau) = \sum_{i=1}^{n} u_i$. The maximum utilization and the maximum and minimum execution cost of any task are denoted $u_{\max}(\tau)$, $e_{\max}(\tau)$, and $e_{\min}(\tau)$, respectively. The task system τ is omitted from this notation when unambiguous.

Soft real-time model. In *soft* real-time systems, tasks may miss their deadlines. This paper is concerned with the derivation of a lateness or *tardiness* [33] bound for a soft, sporadic real-time task system scheduled under EDF-fm (described in Section 4). Formally, the tardiness of a job $\tau_{i,j}$ in schedule $\mathcal{S}$ is defined as *tardiness*$(\tau_{i,j}, \mathcal{S}) = \max(0, t - d_{i,j})$, where t is the time at which $\tau_{i,j}$ completes executing in $\mathcal{S}$. The tardiness of a task system τ under scheduling algorithm $\mathcal{A}$ is defined as the maximum tardiness of any job of any task in τ in any schedule for τ under $\mathcal{A}$. If $\kappa(M)$ is the maximum tardiness of any task system with $U_{sum} \leq M$ under $\mathcal{A}$ on M processors, then $\mathcal{A}$ is said to *ensure a tardiness bound of* $\kappa(M)$ on M processors. Because each task is sequential and jobs have an implicit precedence relationship, a later job cannot commence execution until all prior jobs of the same task have completed execution. Thus, a missed deadline effectively reduces the interval over which the next job should be scheduled in order to meet its deadline.

Though tasks in a soft real-time system are allowed to have nonzero tardiness, we assume that *missed deadlines do not delay future job releases*. Hence, guaranteeing a reasonable bound on tardiness that is independent of time is sufficient to ensure that in the long run each task is allocated a processor share that is in accordance with its utilization. Thus, this model should be useful in settings where maintaining correct share allocations is more important than meeting every deadline. In addition, schemes that ensure bounded tardiness are useful in systems in which a utility function is defined for each task [21]. Such a function specifies the "value" or usefulness of the current job as a function of time; beyond a job's deadline, its usefulness typically decays from a positive value to 0 or below. The amount of time after its deadline beyond which the completion of a job has no value implicitly specifies a tardiness threshold for the corresponding task.

Discussion. Systems that track people and machines, virtual-reality systems, multimedia systems, systems that host web-sites, and some signal-processing systems are some example applications for which the soft real-time model described above may be applicable. For instance, consider a video-decoding task in a multimedia system that is decoding a video stream at the rate of 30 frames per second. While it is desirable that the task decode every frame within 33.3 ms, a tardiness of a few milliseconds will not compromise the quality of the output if the rate of decoding is still 30 frames per second over reasonably long intervals of time. Tardiness may add to jitter in job completion times, but it is unlikely that a jitter of the order of a few tens of milliseconds will be perceptible to the human eye. Similarly, tardiness will add to the buffering needs of a task, but should be reasonable, if the maximum tardiness is reasonably bounded and a system designer is able to choose a tardiness value that balances the processing and memory needs of the system. A similar reasoning can be applied to see that the other example soft real-time applications mentioned above can also tolerate a small, constant tardiness that is independent of elapsed time.

3 Pfair Scheduling Basics

In this section, we describe some basic concepts of Pfair scheduling that are used in the design of EDF-fm.

Currently, Pfair scheduling [11] is the only known way of *optimally* scheduling recurrent (*i.e.*, periodic, sporadic, and rate-based) real-time task systems on multiprocessors. In Pfair scheduling terminology, each task T has an integer execution cost $T.e$ and an integer period $T.p \geq T.e$. The utilization of T, $T.e/T.p$, is also referred to as the *weight* of T and is denoted $wt(T)$. (Note that in the context of Pfair scheduling, tasks are denoted using upper-case letters without subscripts.)

Pfair algorithms allocate processor time in discrete quanta that are uniform in size. Assuming that a quantum is one time unit in duration, the interval $[t, t+1)$, where t is a non-negative integer, is referred to as *slot* t. At most one task may execute on each processor in each slot, and each task may execute on at most one processor in every slot. The sequence of allocation decisions over time slots defines a *schedule* $\mathcal{S}$. Formally, $\mathcal{S} : \tau \times \mathbb{N} \mapsto \{0, 1\}$.

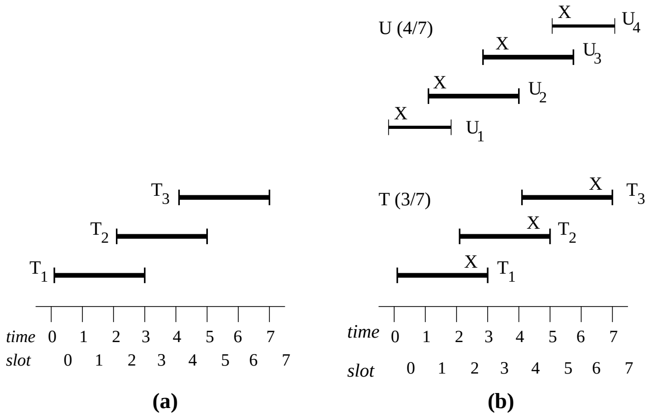


Figure 1: **(a)** Windows of the first job of a periodic task T with weight $3/7$. This job consists of subtasks T_1, T_2, and T_3, each of which must be scheduled within its window. (This pattern repeats for every job.) **(b)** A partial complementary Pfair schedule for a pair of complementary tasks, T and U, on one processor. The slot in which a subtask is scheduled is indicated by an "X." In this schedule, every subtask of U is scheduled in the first slot of its window, while every subtask of T is scheduled in the last slot.

$\mathcal{S}(T,t) = 1$ iff T is scheduled in slot t.

The notion of a Pfair schedule for a periodic[1] task T is defined by comparing such a schedule to an ideal fluid schedule, which allocates $wt(T)$ processor time to T in each slot. Deviation from the allocation in a fluid schedule is captured by the concept of *lag*. Formally, the *lag of task T at time t* in schedule $\mathcal{S}$ is the difference between the total allocations to T in a fluid schedule and $\mathcal{S}$ in the interval $[0,t)$, *i.e.*,

$$lag(T,t,\mathcal{S}) = wt(T) \cdot t - \sum_{u=0}^{t-1} \mathcal{S}(T,u). \tag{1}$$

A schedule $\mathcal{S}$ is said to be *Pfair* iff

$$(\forall T, t :: -1 < lag(T,t,\mathcal{S}) < 1) \tag{2}$$

holds. Informally, the allocation error associated with each task must always be less than one quantum. If relative deadlines are equal to periods, then the above constraints on *lag* are sufficient to ensure that all job deadlines of a periodic task system are met.

The lag constraints above also have the effect of breaking each task T into a potentially infinite sequence of *subtasks*, each with an execution requirement of one quantum. The i^{th} subtask of T is denoted T_i, where $i \geq 1$. Each subtask T_i is associated with a *pseudo-release* $r(T_i)$ and a *pseudo-deadline* $d(T_i)$ defined as follows.

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

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

[1]Unless otherwise specified, by periodic, we mean synchronous, periodic.

To satisfy (2), T_i must be scheduled in the interval $w(T_i) = [r(T_i), d(T_i))$, termed its *window*. Figure 1(a) shows the windows of the first job of a periodic task with weight 3/7. In this example, $r(T_1) = 0$, $d(T_1) = 3$, and $w(T_1) = [0, 3)$ hold.

As mentioned in Section 1, under EDF-fm, each migrating task executes on two processors. To guide us in the assignment of the jobs of a migrating task to its processors (refer Section 4), we define the notion of a *complementary task*.

Definition 1: Task T is said to be *complementary* to U iff $wt(U) = 1 - wt(T)$.

Tasks T and U shown in Figure 1(b) are complementary to one another. A partial Pfair schedule for these two tasks on one processor, in which the subtasks of T are always scheduled in the last slots of their windows and those of U in the first slots, is also shown. We call such a schedule a *complementary schedule*. By Lemma 1 below (proved in an appendix), such a schedule is always possible for two complementary periodic tasks.

Lemma 1 *For any two synchronous, periodic tasks T and U that are complementary, a schedule in which every subtask of T is scheduled in the first slot of its window and every subtask of U in its last slot, or vice versa, is feasible on one processor.*

4 Algorithm EDF-fm

In this section, we present Algorithm EDF-fm (fm denotes that each task is either *fixed* or *migrating*), an EDF-based multiprocessor scheduling algorithm for soft, sporadic real-time task systems. EDF-fm requires no restrictions on total system utilization and can guarantee bounded tardiness for task systems in which each task is light. Because a light task can consume up to half the capacity of a single processor, we do not expect this limitation to be too restrictive in practice. Further, at most $M - 1$ tasks need to be able to migrate, and each such task migrates between two processors and at job boundaries only. This has the benefit of lowering the number of tasks whose states need to be stored on any given processor and the number of processors on which each task's state needs to be stored. Also, the run-time context of a job, which can be expected to be larger than that of a task, need not be transferred between processors.

EDF-fm consists of two phases: an *assignment phase* and an *execution phase*. The assignment phase executes offline and consists of sequentially assigning each task to one or two processors. In the execution phase, jobs are scheduled for execution at run-time such that over reasonable intervals (as explained later), each task executes at a rate that is commensurate with its utilization. The two phases are explained in detail below. The following notation shall be used.

$$s_{i,j} \stackrel{\text{def}}{=} \text{Percentage of } P_j\text{'s processing capacity (expressed as a fraction) allocated to } \tau_i,\ 1 \leq i \leq N, 1 \leq j \leq M. \ (\tau_i \text{ is said to have a } share \text{ of } s_{i,j} \text{ on } P_j.) \tag{5}$$

$$f_{i,j} \stackrel{\text{def}}{=} \frac{s_{i,j}}{u_i}, \text{ the fraction of } \tau_i\text{'s total workload that } P_j \text{ can handle}, 1 \leq i \leq N, 1 \leq j \leq M. \tag{6}$$

$$\rho_i \stackrel{\text{def}}{=} \text{Maximum percentage of } P_i\text{'s processing capacity (expressed as a fraction) that can be allocated to tasks in } \tau,\ 1 \leq i \leq M. \text{ (In other words, the sum of all shares assigned to tasks on } P_i \text{ may not exceed } \rho_i.)$$

4.1 Assignment Phase

The assignment phase allocates or assigns tasks to processors, with each task assigned to either one or two processors. Tasks assigned to two processors are called *migrating* tasks, while those assigned to only one processor are called *fixed* or *non-migrating* tasks. A fixed task τ_i is assigned a *share*, $s_{i,j}$, equal to its utilization u_i on the only processor P_j to which it is assigned. A migrating task has shares on both processors to which it is assigned. The sum of its shares equals its utilization. The assignment phase of EDF-fm also ensures that at most two migrating tasks are assigned to each processor, and that on each P_i, the sum of allocations to all tasks does not exceed a fraction $\rho_i \leq 1$ of P_i's processing capacity. (Since tardiness can be lowered by lowering ρ_i, a value less than one may sometimes be desirable.)

In Figure 2, pseudo-code is given for a a task-assignment algorithm, denoted Assign-Tasks, that satisfies the following properties for every task system τ with $u_{\max}(\tau) \leq \min_{1 \leq i \leq M} \rho_i$ and $U_{sum}(\tau) \leq \sum_{i=1}^{M} \rho_i$.

(P1) Each task is assigned shares on at most two processors. A task's total share equals its utilization.

(P2) Each processor is assigned at most two migrating tasks and may be assigned any number of fixed tasks.

(P3) The sum of the shares allocated to all tasks on Processor P_i is at most ρ_i.

In this pseudo-code, the i^{th} element $u[i]$ of the global array u represents the utilization u_i of task τ_i, $s[i][j]$ denotes $s_{i,j}$ (as defined in (5)), the i^{th} element of array p, which is array $p[i]$, contains the processor(s) to which task i is assigned; arrays $m[i]$ and $f[i]$ denote the migrating tasks and fixed tasks assigned to Processor i, respectively. Note that $p[i]$ and $m[i]$ are each vectors of size two.

Assign-Tasks allocates tasks in sequence to processors, starting from the first processor. Tasks and processors are both considered sequentially. Local variables *proc* and *task* denote the current processor and task, respectively. Tasks are assigned to *proc* as long as the upper limit, ρ_{proc}, on the processing capacity of *proc* is not exhausted. If the current task *task* cannot receive its full share of u_{task} from *proc*, then part of the processing capacity that it requires is allocated on the next processor, *proc* + 1, such that the sum of the shares allocated to *task* on the two processors equals u_{task}. Note that if $u_{\max} \leq \rho_i$, for all i, such an assignment is possible for any task. It is easy to

```
global var
    u : array [1..N] of rational assigned task utilizations;
    ρ:  array [1..M] of rational assigned processor capacities;
    s : array [1..N] of array [1..M] of rational initially 0.0;
    p : array [1..N] of array [1..2] of 0..M initially 0;
    m: array [1..M] of array [1..2] of 0..N initially 0;
    f: array [1..M] of array [1..N] of 0..N initially 0

Algorithm Assign-Tasks()
    local var
    ▷ identifier for current processor
        proc : 1..M initially 1;

    ▷ identifier for current task
        task : 1..N;
    ▷ unassigned utilization on current processor
        AvailUtil : rational;

    ▷ index of the migrating and fixed tasks on current processor
    ▷ mt and ft are used to index into m[proc] and f[proc]
        mt, ft : integer initially 0

    AvailUtil := ρ[1];
    for task := 1 to N do
        if AvailUtil ≥ u[task] then
            s[task][proc] := u[task];
            AvailUtil:= AvailUtil − u[task];
            ft := ft + 1;
            p[task][1] := proc;
            f[proc][ft] := task
        else
            if AvailUtil > 0 then
                    s[task][proc] := AvailUtil;
                    mt := mt + 1;
                    m[proc][mt] := task;
                    p[task][1], p[task][2] := proc, proc + 1;
                    mt, ft := 1, 0;
                    m[proc + 1][mt] := task
            else
                    mt, ft := 0, 1;
                    p[task][1] := proc + 1;
                    f[proc + 1][ft] := task
            fi
            proc := proc + 1;
            s[task][proc] := u[task] − s[task][proc − 1];
            AvailUtil := ρ[proc] − s[task][proc]
        fi
    od
```

Figure 2: Algorithm Assign-Tasks.

see that if $U_{sum} \leq \sum_{i=1}^{M} \rho_i$ also holds, then assigning tasks to processors following this simple approach satisfies (P1)–(P3).

Example 1. Consider a task set τ composed of nine tasks: $\tau_1(5, 20)$, $\tau_2(3, 10)$, $\tau_3(1, 2)$, $\tau_4(2, 5)$, $\tau_5(2, 5)$, $\tau_6(1, 10)$, $\tau_7(2, 5)$, $\tau_8(7, 20)$, and $\tau_9(3, 10)$. The total utilization of this task set is three. A share assignment produced by Assign-Tasks when $\rho_1 = \rho_2 = \rho_3 = 1.0$ is shown in Figure 3. In this assignment, τ_3 and τ_7 are migrating tasks; the remaining tasks are fixed. τ_3 has a share of $\frac{9}{20}$ on processor P_1 and a share of $\frac{1}{20}$ on processor P_2, while τ_7 has shares of $\frac{1}{20}$ and $\frac{7}{20}$ on processors P_2 and P_3, respectively.

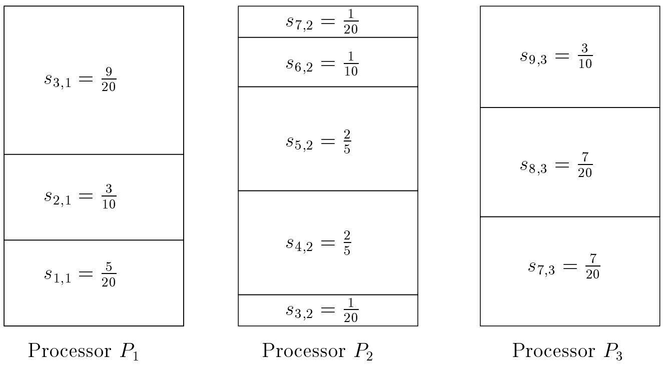


Figure 3: Task assignment on three processors for the task set in Example 1 using Algorithm Assign-Tasks.

4.2 Execution Phase

Having devised a way of assigning tasks to processors, the next step is to design an online scheduling algorithm that is fairly simple, easy to analyze, and can ensure bounded tardiness. For a fixed task, we merely need to decide when to schedule each of its jobs on its (only) assigned processor. For a migrating task, we must decide both *when* and *where* its jobs should execute. Before describing our scheduling algorithm, we discuss some considerations that led to its design.

Design complications. In order to analyze a scheduling algorithm and for the algorithm to guarantee bounded tardiness, it should be possible to bound the total *demand* for execution time by all tasks on each processor over well-defined time intervals. We first argue that bounding total demand may not be possible if the jobs of migrating tasks are allowed to miss their deadlines.

Recall that a deadline miss of a job does not lead to a postponement of the release times of subsequent jobs of the same task. Furthermore, no two jobs of a task may execute in parallel. Hence, the tardiness of a job of a migrating task executing on one processor can postpone the execution of its successor job, which may otherwise execute in a timely manner on a second processor. In the worst case, the second processor may be forced to idle. The tardiness of the second job may also impact the timeliness of fixed tasks and other migrating tasks assigned to the same processor, which in turn may lead to deadline misses of both fixed and migrating tasks on other processors or unnecessary idling on other processors.

As a result, a set of dependencies is created among the jobs of migrating tasks, resulting in an intricate linkage among processors that complicates scheduling analysis. It is unclear how per-processor demand can be precisely bounded when activities on different processors become interlinked.

Let us look at a concrete example that reveals this linkage among processors. Consider the task set τ, introduced earlier, with task assignments and processor shares shown in Figure 3. For simplicity, assume that the execution of the jobs of a migrating task alternate between the two processors to which the task is assigned. τ_3 releases its first job on P_1, while τ_7 releases its first job on P_3. (We are assuming such a naïve assignment pattern to illustrate the processor linkage using a short segment of a real schedule. Such a linkage occurs even with an intelligent job-assignment pattern if migrating tasks miss their deadlines.) A complete schedule up to time 27, with the jobs assigned to each processor scheduled using EDF, is shown in Figure 4.

In Figure 4, the sixth job of the migrating task τ_3 misses its deadline (at time 12) on P_2 and completes executing at time 14. This prevents the next job of τ_3 released on P_1 from being scheduled until time 14 and it misses its deadline. Because job releases are not postponed due to deadline misses, the seventh job of τ_3 is released at time 12 and has a deadline at time 14.

The missed deadline of the migrating task τ_3 impacts the execution of the fixed tasks also on P_2. (It may seem that τ_3's misses can be avoided by determining processor assignments for its jobs dynamically. However, a reasonable strategy that is not convoluted does not appear to be possible.) The deadline misses of the fixed tasks τ_4, τ_5, and τ_6 cause the migrating task τ_7 to miss a deadline on P_2. In particular, the fourth job of τ_7 misses its

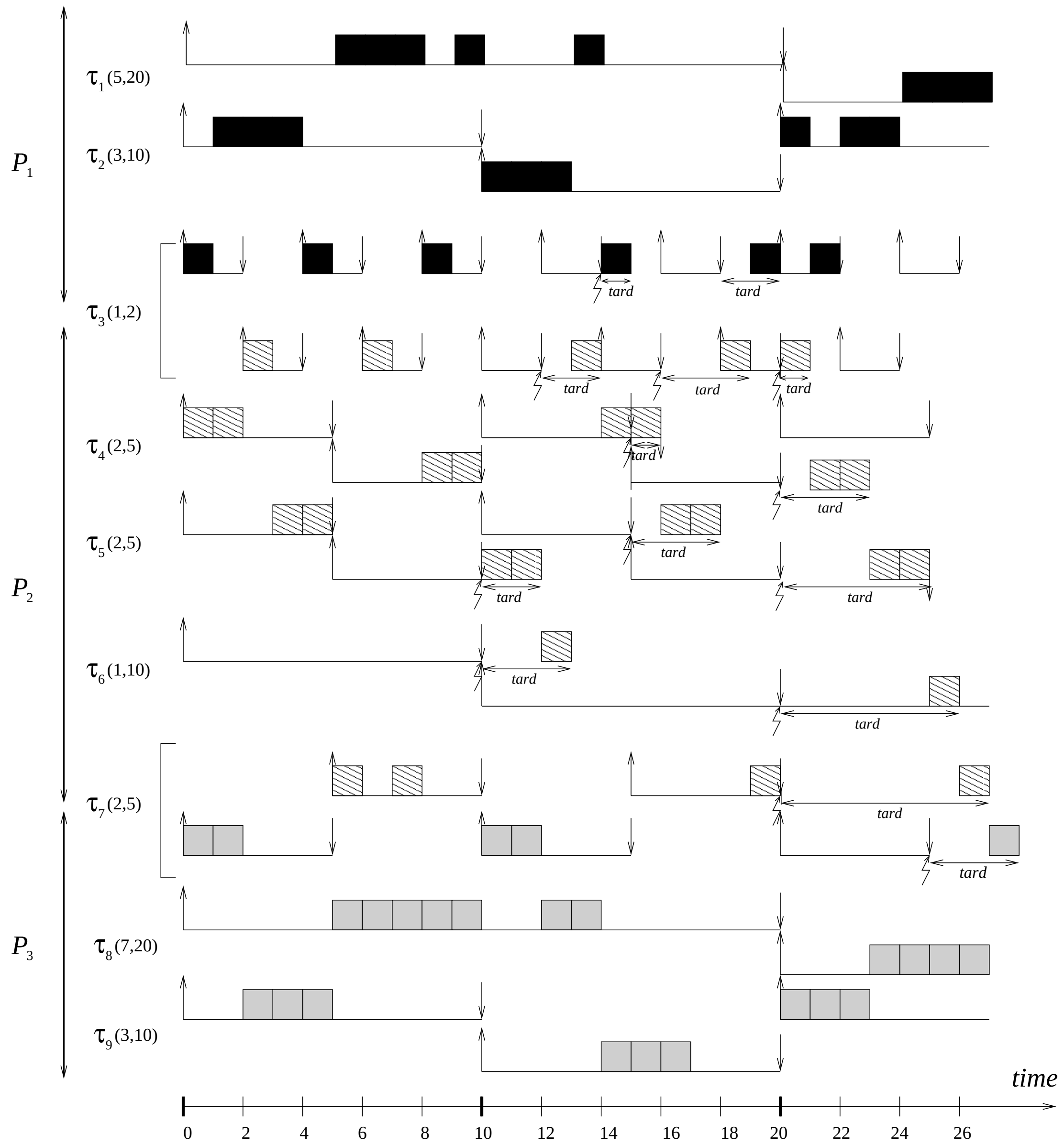


Figure 4: Illustration of processor linkage.

deadline, which in turn reduces the interval over which the fifth job of the same task can execute on P_3. Note that on P_3, during [25,27), the fifth job of τ_7 is waiting (while the second job of τ_8 with a later deadline is scheduled) due to its dependence on its prior job, which is still pending on P_2. Thus, a nontrivial linkage is established among the processors that impacts system tardiness.

In order to eliminate the processor linkage described above that complicates analysis, choosing to ensure that *migrating tasks never miss their deadlines* was unavoidable in the design of EDF-fm. Consequently, the scheduling algorithm deployed on each processor and the guidelines used in distributing jobs of migrating tasks are as described in the following subsections.

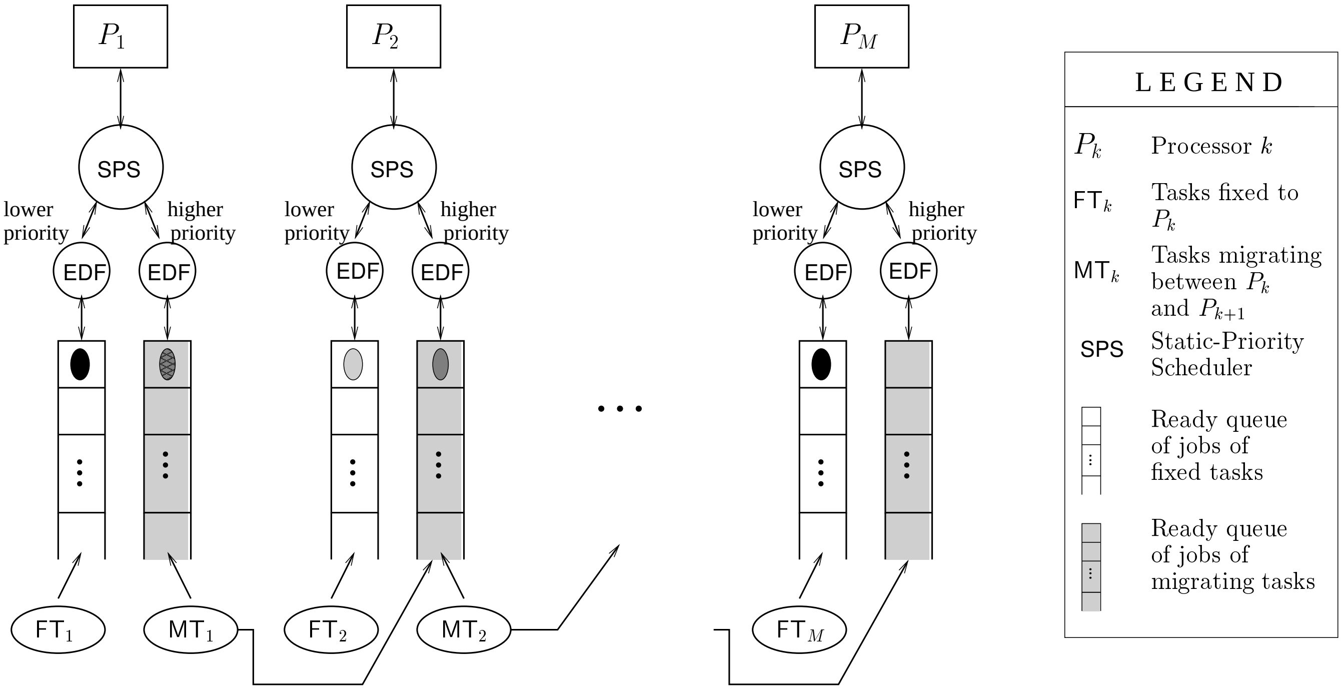


Figure 5: Schematic representation of EDF-fm in the execution phase.

4.2.1 Per-Processor Scheduling Rules

In this subsection, we describe how each processor schedules its jobs.

As shall be seen in the next subsection, jobs of migrating tasks are assigned to processors using static rules that are independent of run-time dynamics. The jobs assigned to a processor are scheduled independently of other processors, and on each processor, migrating tasks are statically prioritized over fixed tasks. Jobs within each task class (*i.e.*, fixed and migrating) are scheduled using EDF, which is optimal on uniprocessors. A schematic representation of EDF-fm in the execution phase is shown in Figure 5. (Practical considerations in implementing EDF-fm in a real system are discussed at the end of this section after the algorithm has been described in its entirety.) The per-processor priority scheme described above, together with the restriction that migrating tasks have utilizations at most 1/2, and the task assignment property (from (P2)) that there are at most two migrating tasks per processor, ensures that migrating tasks never miss their deadlines. To see this, first observe that the execution of jobs of migrating tasks cannot be impacted by fixed tasks (because migrating tasks are accorded higher priority than fixed tasks). Next, the sum of the utilizations of the up to two migrating tasks assigned to any processor is at most one. Hence, since EDF is the second level scheduler, regardless of how a migrating task's jobs are distributed between its processors, all deadlines of migrating tasks will be met. Therefore, the jobs of migrating tasks executing on different processors do not impact one another, and each processor can be analyzed independently. Thus, the multiprocessor scheduling analysis problem at hand is transformed into a simpler

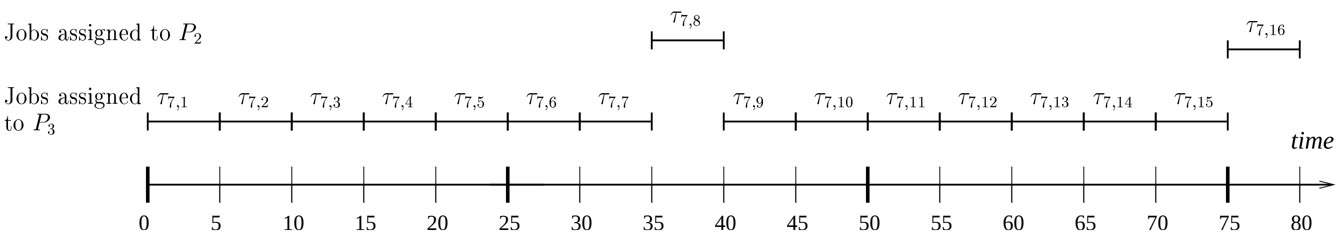


Figure 6: Assignment of periodically released jobs of migrating task τ_7 to processors P_2 and P_3.

uniprocessor one.

4.2.2 Distribution of Jobs of Migrating Tasks

In the description of EDF-fm, we are left with defining static rules that map the jobs of migrating tasks to processors. Since a migrating task will never miss deadlines under the scheduling algorithm described in Section 4.2.1 above, regardless of how its jobs are distributed, our main goal in the design of the distribution algorithm is to minimize tardiness for fixed tasks.

Considerations in the distribution of migrating jobs. A naïve assignment of the jobs of a migrating task to its processors can cause an over-utilization on one of its assigned processors and adversely impact fixed tasks. To lower tardiness for fixed tasks, our goal is to determine a job distribution pattern that can prevent over-utilization in the long run by ensuring that over well-defined time intervals (explained later), on each processor, the execution of a migrating task is roughly uniform and the demand the task places is in accordance with its allocated share on that processor.

For example, consider the migrating task $\tau_7(2,5)$ in Example 1. τ_7 has a share of $s_{7,2} = \frac{1}{20}$ on P_2 and $s_{7,3} = \frac{7}{20}$ on P_3. Also, $f_{7,2} = \frac{s_{7,2}}{u_7} = \frac{1}{8}$ and $f_{7,3} = \frac{s_{7,3}}{u_7} = \frac{7}{8}$, which imply that P_2 and P_3 should be capable of executing $\frac{1}{8}$ and $\frac{7}{8}$ of the workload of τ_7, respectively. Our goal is to devise a job assignment pattern that would ensure that, in the long run, the fraction of a migrating task τ_i's workload executed on P_j is close to $f_{i,j}$, and at any time deviation from this ideal is minimized. One such job assignment pattern for τ_7 over interval $[0, 80)$ is shown in Figure 6. Assuming that τ_7 is a synchronous, periodic task,[2] the pattern in $[0, 40)$ would repeat every 40 time units.

In the job assignment of Figure 6, exactly one job out of every eight consecutive jobs of τ_7 released in the interval $[5k, 5(k+8))$, where $k \geq 0$, is assigned to P_2. Because $e_7 = 2$, τ_7 places a demand for two units of time, *i.e.*, $1/20^{\text{th}}$ of P_2, in $[5k, 5(k+8))$. Because τ_7 is allocated a share of $s_{7,2} = 1/20$ on P_2, this job assignment pattern ensures that in the long run τ_7 does not overload P_2. However, the demand due to τ_7 on P_2 over short intervals may exceed or fall below the share allocated to it. For instance, with this assignment, τ_7 requires two units of time, *i.e.*, 2/5 of

[2] The first job of a synchronous, periodic task is released at time 0.

P_2 in the interval $[40k+35, 40(k+1))$, and zero time units in the interval $[40k, 40k+35)$. Similarly, exactly seven out of every eight consecutive jobs of τ_7 are assigned to P_3. Thus, τ_7 requires 14 units of time, or 7/20 of the time, in $[5k, 5(k+8))$, and the share allocated to it matches this need. However, as with P_2, the demand due to τ_7 on P_3 over shorter intervals may deviate from its long-term share.

A job assignment pattern like the one described above can ensure that, over the long term, the demand of each migrating task on each processor is in accordance with the share allocated to it. However, as explained above, such an assignment pattern can result in a migrating task overloading a processor over short time intervals, leading to deadline misses for fixed tasks. Nevertheless, because a deadline miss of a job does not delay the next job release of the same task, this scheme also ensures, over the long term, that each fixed task executes at its prescribed rate (given by its utilization). Later in this section, we show that the amount by which fixed tasks can miss their deadlines due to the transient overload of migrating tasks is bounded.

A job assignment pattern similar to the one in Figure 6 can be defined for any migrating task. We draw upon some concepts of Pfair scheduling to derive formulas that can be used to determine such a pattern at run-time. As mentioned in Section 3, currently, Pfair scheduling [11] is the only known way of optimally scheduling recurrent real-time task systems on multiprocessors. Pfair algorithms achieve optimality by requiring each task to execute at a more uniform rate, given by its utilization, than mandated by the periodic or the sporadic task models. In fact, the allocation error at any time for optimal Pfair algorithms, in comparison to ideal fluid algorithms that can execute each task at its precise rate, is less than one time unit. It is also known that, in general, an allocation error lower than that guaranteed by Pfair, is not possible in practice [11]. Hence, we consider Pfair scheduling rules to be appropriate for distributing the jobs of migrating tasks. A review of the needed Pfair scheduling concepts is provided in Section 3. (We stress that we are *not* using Pfair algorithms in our scheduling approach. We merely wish to borrow some relevant formulas from the Pfair scheduling literature.)

Distribution rules. Let τ_i be any migrating periodic task (we later relax the assumption that τ_i is periodic) that is assigned shares $s_{i,j}$ and $s_{i,j+1}$ on processors P_j and P_{j+1}, respectively. (Recall that every migrating task is assigned shares on two consecutive processors by ASSIGN-TASKS.) As explained earlier, $f_{i,j}$ and $f_{i,j+1}$ (given by (6)) denote the fraction of the workload (*i.e.*, the total execution requirement) of T that should be executed on P_j and P_{j+1}, respectively, in the long run. By (P1), the total share allocated to τ_i on P_j and P_{j+1} is u_i. Hence, by (6), it follows that

$$f_{i,j} + f_{i,j+1} = 1. \tag{7}$$

Assuming that the execution cost and period of every task are rational numbers (which can be expressed as a ratio of two integers), u_i, $s_{i,j}$, and hence, $f_{i,j}$ and $f_{i,j+1}$ are also rational numbers. Let $f_{i,j} = \frac{x_{i,j}}{y_i}$, where $x_{i,j}$ and y_i

are positive integers that are relatively prime. Then, by (7), it follows that $f_{i,j+1} = \frac{y_i - x_{i,j}}{y_i}$. Therefore, one way of distributing the workload of τ_i between P_j and P_{j+1} that is commensurate with the shares of τ_i on the two processors would be to assign $x_{i,j}$ out of every y_i jobs to P_j and the remaining jobs to P_{j+1}.

Rather than arbitrarily choosing the $x_{i,j}$ jobs to assign to P_j, we borrow from the aforementioned concepts of Pfair scheduling to guide in the distribution of jobs. For illustration, consider a migrating task τ_i with utilization $\frac{3}{8}$ that is assigned shares $s_{i,j} = \frac{1}{5}$ and $s_{i,j+1} = \frac{7}{40}$ on P_j and P_{j+1}, respectively. Hence, $f_{i,j} = \frac{8}{15}$ and $f_{i,j+1} = \frac{7}{15}$ hold. Therefore, one way of distributing τ_i's jobs would be to assign the first eight of jobs $15k+1, \cdots, (15k+15)$, for all $k \geq 0$, to P_j, and the remaining jobs to P_{j+1}. Though such a strategy is reasonable (and perhaps among the best) for the example in Figure 6, the distribution may be significantly uneven over short durations for some task systems, such as the present example, and the transient overload that ensues may be quite excessive. As we shall see, a more even distribution of jobs can be obtained by applying Pfair rules.

If we let two fictitious periodic Pfair tasks V and W correspond to processors P_j and P_{j+1}, respectively, let $f_{i,j}$ and $f_{i,j+1}$ denote their weights, and let a quantum span p_i time units, then the following analogy can be made between the jobs of the migrating task τ_i and the subtasks of the fictitious tasks V and W. First, slot s can be associated with job $s+1$ in that slot s represents the interval in which the $(s+1)^{\text{st}}$ job of τ_i, which is released at the beginning of that slot, needs to be scheduled. (Recall that slots are numbered starting from 0.) This is illustrated in Figure 7(a), which depicts the layouts of subtasks in the first period of V and W for the example mentioned above, and a complementary schedule for those subtasks on a fictitious processor. Refer to "slots" and "jobs" marked beneath the time line. Next, subtask V_g represents the g^{th} job assigned to P_j (of the jobs of τ_i); that is, exactly one of the jobs that correspond to slots $r(V_g), \cdots, d(V_g) - 1$ should be assigned as the g^{th} job of P_j. Similarly, subtask W_h represents the h^{th} job assigned to P_{j+1}. Finally, if subtask V_g (resp., W_h) is scheduled in slot s (on a fictitious processor), then the $(s+1)^{\text{st}}$ job of τ_i should be assigned to P_j (resp., P_{j+1}). In other words, job $s+1$ is assigned to the processor that corresponds to the Pfair task that is scheduled in slot s. Referring to Figure 7(a), since subtask V_1 is scheduled in slot 0, the first job of τ_i is assigned to P_j. Similarly, since subtasks of V are scheduled in slots 1, 3, 5, 7, 9, 11, and 13, jobs 2, 4, 6, 8, 10, 12, and 14 of τ_i are assigned to P_j, and since subtasks of W are scheduled in slots 2, 4, 6, 8, 10, 12, and 14, jobs 3, 5, 7, 9, 11, 13, and 15 of τ_i are assigned to P_{j+1}.

By Definition 1 and (7), Pfair tasks V and W are complementary. Therefore, by Lemma 1, a complementary schedule for V and W in which the subtasks of V are scheduled in the first slot of their windows and those of W in the last slot of their windows is feasible. Further, because $wt(V) + wt(W) = 1$, some subtask is scheduled in each slot. Hence, the following holds.

(A1) Exactly one of the subtasks of V and W is scheduled in each slot.

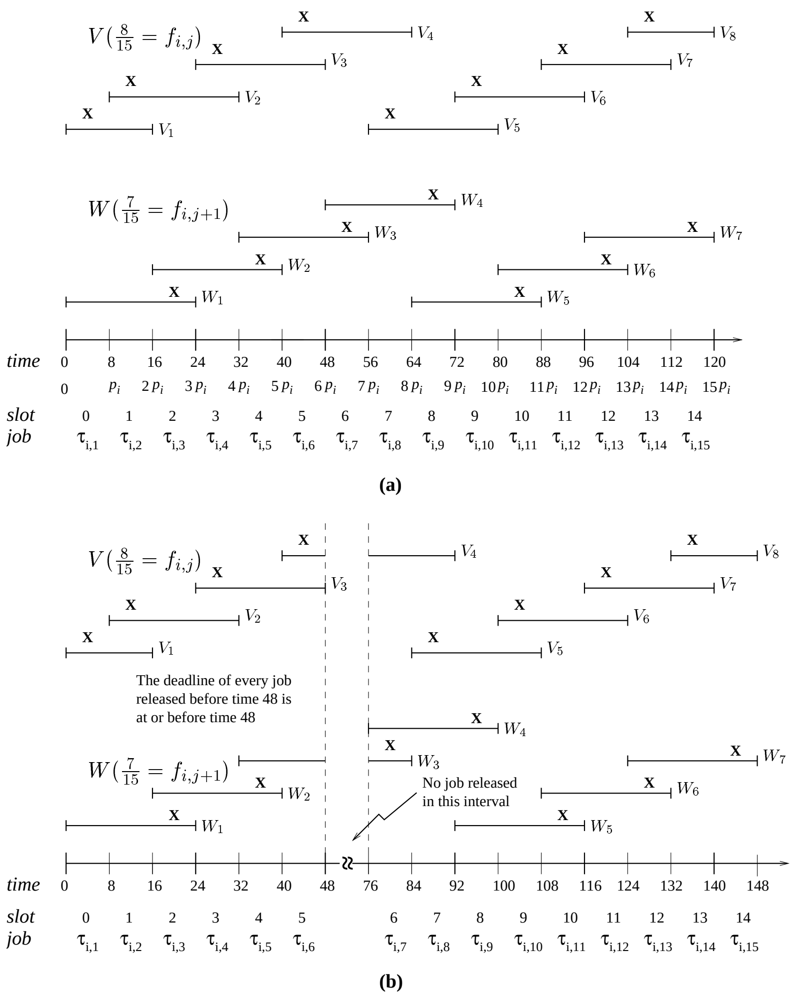


Figure 7: Complementary Pfair schedule for tasks V and W with weights $f_{i,j} = 8/15$ and $f_{i,j+1} = 7/15$, respectively, that guides the assignment of jobs of task $\tau_i(3, 8)$ to processors P_i and P_{j+1}. Subtasks in the first period, $[0, 15)$, of V and W are shown. The pattern repeats for every period. Slot k corresponds to job $k+1$ of τ_i. The slot in which a subtask is scheduled is indicated by an "X." **(a)** The jobs of τ_i are released periodically. **(b)** The seventh job of τ_i is delayed by 28 time units.

Accordingly, we consider a job assignment policy in which the job of τ_i corresponding to the first slot in the window of subtask V_g is assigned as the g^{th} job of τ_i to P_j and the job of τ_i corresponding to the last slot in the window of subtask W_h is assigned as the h^{th} job of τ_i to P_{j+1}, for all g and h. By (A1), this policy satisfies the following property.

(A2) Each job of τ_i is assigned to exactly one of P_j and P_{j+1}.

More generally, we can use the formula for the release time of a subtask given by (3) for job assignments. Let job_i denote the total number of jobs released by task τ_i and that have been assigned, and let $job_{i,j}$ denote the total number of these jobs that have been assigned to P_j. Let $p_{i,\ell}$ denote the processor to which job ℓ of task τ_i is assigned. Then, the processor to which job $job_i + 1$ is assigned is determined as follows.

$$p_{i,job_i+1} = \begin{cases} j, & \text{if } job_i = \left\lfloor \dfrac{job_{i,j}}{f_{i,j}} \right\rfloor \\ j+1, & \text{otherwise} \end{cases} \tag{8}$$

As before, let $f_{i,j}$ and $f_{i,j+1}$ be the weights of two fictitious Pfair tasks V and W, respectively. Then, by (3), $t_r = \left\lfloor \frac{job_{i,j}}{f_{i,j}} \right\rfloor$ denotes the release time of subtask $V_{job_{i,j}+1}$ of task V. Thus, (8) assigns to P_j the job that corresponds to the first slot in the window of subtask V_g as the g^{th} job of τ_i on P_j, for all g. (Recall that the index of the job of the migrating periodic task τ_i that is released in slot t_r is given by $t_r + 1$.) Because the sum of the weights of the two tasks is one, Lemma 1 implies that if t_r is not the release time of any subtask of V, then $t_r + 1$ is the deadline of some subtask of W. Thus, (8) ensures that the job that corresponds to the last slot in the window of subtask W_h is assigned as the h^{th} job of τ_i on P_{j+1}, for all h.

Thus far in our discussion, to simplify the presentation, we have assumed that the job releases of task τ_i are periodic. However, note that the job assignment given by (8) is independent of "real" time and is based on job numbers only. Hence, assigning jobs using (8) should be sufficient to ensure (A2) even when τ_i is sporadic. This is illustrated in Figure 7(b). Here, we assume that τ_i is a sporadic task, whose seventh job release is delayed, by 28 time units, to time 76 from time 48. As far as τ_i is concerned, the interval $[48, 76)$ is "frozen" and the job assignment resumes at time 76. As indicated in the figure, in any such interval in which activity is suspended for a migrating task τ_i, no jobs of τ_i are released. Furthermore, the deadlines of all jobs of τ_i released before the frozen interval fall at or before the beginning of the interval.

We next prove a property that bounds from above the number of jobs of a migrating task assigned to each of its processors by the job assignment rule given by (8).

Lemma 2 *Let τ_i be a migrating task that is assigned to processors P_j and P_{j+1}. The number of jobs out of any consecutive $\ell \geq 0$ jobs of τ_i that are assigned to P_j and P_{j+1} is at most $\lceil \ell \cdot f_{i,j} \rceil$ and $\lceil \ell \cdot f_{i,j+1} \rceil$, respectively.*

Proof: We first prove the lemma for the number of jobs assigned to P_j. We begin by claiming the following.

(J) Exactly $\lceil \ell_0 \cdot f_{i,j} \rceil$ of the first ℓ_0 jobs of τ_i are assigned to P_j.

(J) holds trivially when $\ell_0 = 0$. Therefore, assume $\ell_0 \geq 1$. Let q denote the total number of jobs of the first ℓ_0 jobs of τ_i that are assigned to P_j. (By (8), the first job of τ_i is assigned to P_j, hence, $q \geq 1$ holds.) Then, there exists an $\ell' \leq \ell_0$ such that job ℓ' of τ_i is the q^{th} job of τ_i assigned to P_j. Therefore, by (8),

$$\ell' - 1 = \left\lfloor \frac{q-1}{f_{i,j}} \right\rfloor \tag{9}$$

holds. (Note that job_i of (8) denotes the number of jobs of τ_i that have already been distributed, and hence, is equal to $\ell' - 1$ here. Similarly, $job_{i,j}$ denotes the number of jobs already assigned to P_j, and so is equal to $q-1$.) ℓ, ℓ', and q denote job numbers or counts, and hence are all non-negative integers. By (9), we have

$$\frac{q-1}{f_{i,j}} \geq \ell' - 1 \quad \Rightarrow \quad q-1 \geq (\ell'-1) \cdot f_{i,j} \quad \Rightarrow \quad q > \ell' \cdot f_{i,j} \qquad \text{(because } f_{i,j} < 1\text{)}, \tag{10}$$

and

$$\frac{q-1}{f_{i,j}} < \ell' \quad \Rightarrow \quad q-1 < \ell' \cdot f_{i,j} \quad \Rightarrow \quad q < \ell' \cdot f_{i,j} + 1. \tag{11}$$

Because q is an integer, by (10) and (11), we have

$$q = \lceil \ell' \cdot f_{i,j} \rceil. \tag{12}$$

If $\ell' = \ell_0$ holds, then (J) follows from (12) and our definition of q. On the other hand, to show that (J) holds when $\ell' < \ell_0$, we must show that $q = \lceil \hat{\ell} \cdot f_{i,j} \rceil$ holds for all $\hat{\ell}$, where $\ell' < \hat{\ell} \leq \ell_0$. (Note that $\hat{\ell}$ is an integer.) By the definitions of q, ℓ', and ℓ_0, q of the first ℓ' jobs of τ_i are assigned to P_j, and none of the jobs $\ell'+1$ through ℓ_0 is assigned to P_j. Therefore, by (8), it follows that $\hat{\ell} - 1 < \left\lfloor \frac{q}{f_{i,j}} \right\rfloor$ holds for all $\hat{\ell}$, where $\ell' < \hat{\ell} \leq \ell_0$. (As before, because $\hat{\ell}$ is the index of the next job of τ_i to be distributed, job_i of (8) equals $\hat{\ell} - 1$. However, since the number of jobs already assigned to P_j is q, $job_{i,j} = q$.) Thus, we have the following, for all $\hat{\ell}$, where $\ell' < \hat{\ell} \leq \ell_0$.

$$\left\lfloor \frac{q}{f_{i,j}} \right\rfloor > \hat{\ell} - 1 \Rightarrow \left\lfloor \frac{q}{f_{i,j}} \right\rfloor \geq \hat{\ell} \Rightarrow \frac{q}{f_{i,j}} \geq \hat{\ell} \Rightarrow q \geq \hat{\ell} \cdot f_{i,j} \Rightarrow q \geq \lceil \hat{\ell} \cdot f_{i,j} \rceil \quad \text{(because } q \text{ is an integer)} \tag{13}$$

By (12) and because $\hat{\ell} > \ell'$ holds, (13) implies that $\lceil \hat{\ell} \cdot f_{i,j} \rceil = \lceil \ell' \cdot f_{i,j} \rceil = q$.

To complete the proof for P_j, we show that at most $\lceil \ell \cdot f_{i,j} \rceil$ of any consecutive ℓ jobs of τ_i are assigned to P_j. Let $\mathcal{J}$ represent jobs $\ell_0 + 1$ to $\ell_0 + \ell$ of τ_i, where $\ell_0 \geq 0$. By (J), exactly $\lceil \ell_0 \cdot f_{i,j} \rceil$ of the first ℓ_0 jobs and $\lceil (\ell_0 + \ell) \cdot f_{i,j} \rceil$ of the first $\ell_0 + \ell$ jobs of τ_i are assigned to P_j. Therefore, the number of jobs belonging to $\mathcal{J}$ that are assigned to P_j, denoted $Jobs(\mathcal{J}, j)$, is given by

$$Jobs(\mathcal{J}, j) = \lceil (\ell_0 + \ell) \cdot f_{i,j} \rceil - \lceil \ell_0 \cdot f_{i,j} \rceil \leq \lceil \ell_0 \cdot f_{i,j} \rceil + \lceil \ell \cdot f_{i,j} \rceil - \lceil \ell_0 \cdot f_{i,j} \rceil = \lceil \ell \cdot f_{i,j} \rceil,$$

which proves the lemma for the number of jobs assigned to P_i. (The second step in the above derivation follows from $\lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil$.)

Finally, we are left with proving the lemma for P_{j+1}. By the job assignment rule in (8), every job of τ_i is assigned to exactly one of P_j and P_{j+1}. Therefore (J) implies that exactly $\ell_0 - \lceil \ell_0 \cdot f_{i,j} \rceil$ of the first ℓ_0 jobs of τ_i

are assigned to P_{j+1}. Hence, the number of jobs belonging to $\mathcal{J}$ that are assigned to P_{j+1} is given by

$$
\begin{aligned}
Jobs(\mathcal{J}, j+1) &= (\ell_0 + \ell) - \lceil (\ell_0 + \ell) \cdot f_{i,j} \rceil - \ell_0 + \lceil \ell_0 \cdot f_{i,j} \rceil \\
&= \ell - \lceil (\ell_0 + \ell) \cdot f_{i,j} \rceil + \lceil \ell_0 \cdot f_{i,j} \rceil \\
&\leq \ell - \lceil \ell_0 \cdot f_{i,j} \rceil - \lfloor \ell \cdot f_{i,j} \rfloor + \lceil \ell_0 \cdot f_{i,j} \rceil \qquad \text{(because } \lceil x+y \rceil \geq \lceil x \rceil + \lfloor y \rfloor) \\
&= \ell - \lfloor \ell \cdot f_{i,j} \rfloor \\
&< \ell - \ell \cdot f_{i,j} + 1 \\
&= \ell \cdot f_{i,j+1} + 1 \qquad \text{(by (7)).}
\end{aligned}
$$

Because $Jobs(\mathcal{J}, j+1)$ is an integer, the above implies that $Jobs(\mathcal{J}, j+1) \leq \lceil \ell \cdot f_{i,j+1} \rceil$, completing the proof. ∎

4.3 Illustration of Distribution and Scheduling

In this subsection, we illustrate the distribution and scheduling algorithms described in Sections 4.2.2 and 4.2.1, respectively, by applying them to the following task set.

Example 2. Let τ be a task set with the following eight tasks: $\tau_1(9, 20)$, τ_2–$\tau_7(3, 8)$, and $\tau_8(3, 10)$. The total utilization of this task set is 3.0 and an assignment of these tasks to three processors by Algorithm ASSIGN-TASKS is shown in Figure 8. In this assignment, tasks τ_3 and τ_6 are migrating, and the shaded blocks denote the shares assigned to these tasks.

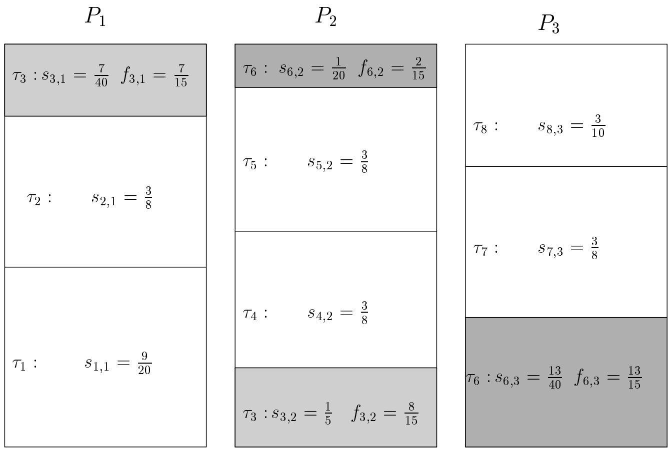


Figure 8: Assignment of tasks in Example 2 to three processors.

Based on the shares allotted to the migrating tasks τ_3 and τ_6 on their respective processors, (8) can be used to determine how their jobs are distributed. For example, let us consider τ_3. As in Section 4.2.2, let job_i denote the total number of jobs of τ_i that have been released and already assigned to some processor, and let $job_{i,j}$ denote the number of these jobs assigned to processor P_j. Letting P_1 (resp., P_2) denote processor P_j (resp., P_{j+1}), to begin with, we have $job_3 = job_{3,1} = job_{3,2} = 0$. Hence, by (8), since $\left\lfloor \dfrac{job_{3,1}}{f_{3,1}} \right\rfloor = \left\lfloor \dfrac{0}{7/15} \right\rfloor = 0 = job_3$, the job numbered $job_i + 1 = 0 + 1 = 1$ is assigned to P_1. After

Task τ_3		Task τ_6	
Job No.	Assigned To	Job No.	Assigned To
1	P_1	1	P_2
2	P_2	2	P_3
3	P_1	3	P_3
4	P_2	4	P_3
5	P_1	5	P_3
6	P_2	6	P_3
7	P_1	7	P_3
8	P_2	8	P_2
9	P_1	9	P_3
10	P_2	10	P_3
11	P_1	11	P_3
12	P_2	12	P_3
13	P_1	13	P_3
14	P_2	14	P_3
15	P_2	15	P_3

Table 2: Distribution of jobs of migrating tasks in Example 2.

this assignment, we have $job_{3,1} = job_3 = 1$, and hence, $\left\lfloor \frac{job_{3,1}}{f_{3,1}} \right\rfloor = \left\lfloor \frac{1}{7/15} \right\rfloor = 2 \neq job_3$. Consequently, by (8), the second job is assigned to P_2. The assignment of jobs to processors obtained by following this algorithm for the first 15 jobs of the two migrating tasks is provided in Table 2.

Each processor schedules the jobs of its fixed tasks and those of migrating tasks that are assigned to it independently, as described in Section 4.2.1. A segment of such a schedule for the task set in this example (assuming periodic job releases) is provided in Figure 9. In this figure, the dashed lines demarcate the jobs assigned to different processors, and the jobs immediately adjacent to each line on either side belong to a single migrating task. Observe that migrating tasks are accorded higher priority than fixed tasks.

Comparison to other similar approaches. The striping of task utilizations to processors used in our task assignment algorithm bears similarities to how task executions are striped within a "time block" in Algorithms SA1 and SA2 proposed in [22] for scheduling hard real-time tasks. In these algorithms, processor time is considered in chunks of quanta referred to as "blocks." All the blocks are equally sized, with the common size given by the greatest common divisor (GCD) of the periods of all the tasks. The number of quanta to be allocated to each task in each block is computed based on task utilizations and a schedule for the various tasks that is common across blocks is laid out offline. The block schedule is constructed by simply striping task executions sequentially across processors, just as processor shares are allocated to tasks in Assign-Tasks. As in Assign-Tasks, at most $M - 1$ tasks migrate between two processors, but within each block. Since the size of a block is at most the minimum task period, each job of such a task migrates at least once. Furthermore, each fixed task is preempted once in each

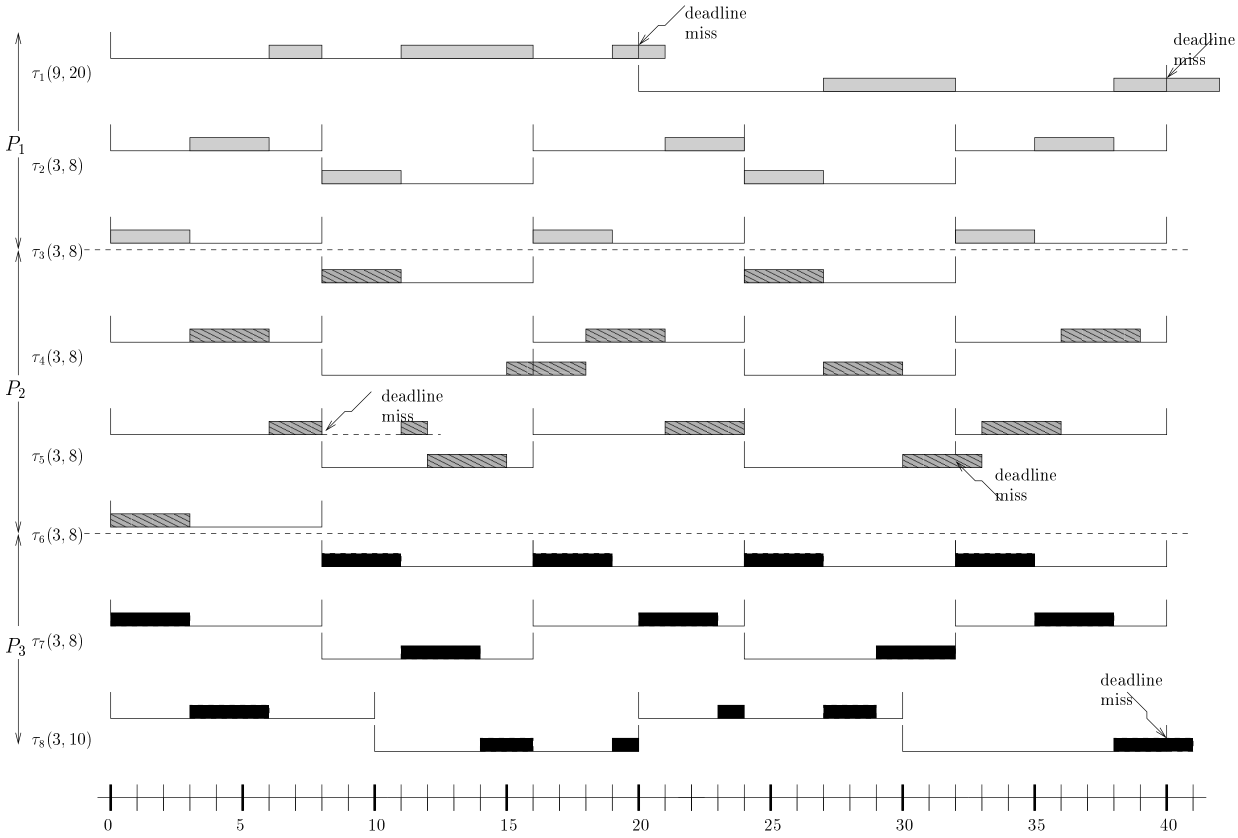


Figure 9: An initial segment of a schedule under EDF-fm for the task set in Example 2.

block. (In contrast, under EDF-fm, no migrating job migrates, and the total number of job preemptions is at most the total number of jobs.) It is worth noting that this approach attempts to discretize and improve the practicality of processor sharing schedules [14], in which each task is allocated its share within each quantum, which can be infinitesimal. The approach nevertheless suffers from some associated drawbacks, such as higher preemption and migration overheads, and is quite brittle and can support only periodic tasks (since block layouts and schedules will not be valid if jobs are delayed). Moreover, since practical considerations require that the number of quanta allocated to each task in a block be integral, block schedules are not feasible for all task systems, but only to those whose task parameters satisfy certain constraints. As can be seen, one predominant difference between our approach and that in the SA algorithms is that in the latter, all the tasks, and in particular migrating tasks, are allocated processor time more uniformly, which improves schedulability for certain task sets, albeit at the expense of decreased flexibility and increased overheads.

4.4 Practical Implementation

In this subsection, we briefly discuss how EDF-fm can be implemented in a real system, and argue that it incurs very little additional overhead than partitioning. This is mainly because the processor to which a migrating task's job is assigned can be determined solely based on the job index and migrating tasks do not miss deadlines. Hence, if a migrating task is periodic, each of its processor schedulers can easily determine the release time of the next job to be assigned to it, and can enqueue that job in the *release queue* associated with its release time.[3] (From the description in Section 4.2.2, it follows that if k jobs of a migrating task τ_i have already been assigned to processor P_j, then the release time of the $(k+1)^{\text{st}}$ job to be assigned to it is given by either $\left\lfloor \frac{k}{f_{i,j}} \right\rfloor \cdot p_i$ or $(\left\lceil \frac{k+1}{1-f_{i,j}} \right\rceil - 1) \cdot p_i$.) On the other hand, if a migrating task is sporadic, then each of its job arrivals can be programmed to trigger interrupts on both of its processors. Here again, since the processor on which the job executes depends on the job index, one of the processors can simply ignore arrival interrupts that are not meant for it. Further, the two logical *ready queues*[4] (one each for the ready jobs of fixed and migrating tasks) associated with each processor (depicted in Figure 5), can be implemented using a single physical ready queue with a bit to denote whether a job belongs to a fixed task. Thus, implementing EDF-fm incurs very little overhead, if any, in comparison to partitioned-EDF.

We are now ready to derive a tardiness bound for EDF-fm.

5 Tardiness Bound for EDF-fm

As discussed earlier, jobs of migrating tasks do not miss their deadlines under EDF-fm. Also, if no migrating task is assigned to processor P_k, then the fixed tasks on P_k do not miss their deadlines. Hence, our analysis is reduced to determining the maximum amount by which a job of a fixed task may miss its deadline on each processor P_k in the presence of migrating jobs. We assume that two migrating tasks, denoted τ_i and τ_j, are assigned to P_k. (A tardiness bound with only one migrating task can be deduced from that obtained with two migrating tasks.) We prove the following.

(L) The tardiness of a fixed task τ_q assigned to P_k is at most $\Delta = \frac{e_i(f_{i,k}+1)+e_j(f_{j,k}+1)-p_q(1-\rho_k)}{1-s_{i,k}-s_{j,k}}$.

The proof is by contradiction. Contrary to (L), assume that job $\tau_{q,\ell}$ of a fixed task τ_q assigned to P_k has a tardiness exceeding Δ. We use the following notation to assist with our analysis. System start time is taken to be zero and the processor is assumed to be idle before time zero.

$$t_d \stackrel{\text{def}}{=} \text{absolute deadline of job } \tau_{q,\ell} \tag{14}$$

$$t_c \stackrel{\text{def}}{=} t_d + \Delta \tag{15}$$

$$t_0 \stackrel{\text{def}}{=} \begin{array}{l}\text{latest instant before } t_c \text{ such that no migrating job released before} \\ t_0 \text{ and assigned to } P_k \text{ or a fixed job released before } t_0 \text{ with deadline} \\ \text{at most } t_d \text{ is pending at } t_0\end{array} \tag{16}$$

[3] A time tick's *release queue* is a *priority queue* of all the jobs that are to be released at that time. A release queue is merged with the ready queue at the queue's release time.
[4] A *ready queue* is a priority queue of ready jobs.

Note that by the definition of t_0, P_k either is idle or executes a job of a fixed task with deadline later than t_d at $t_0 - \epsilon$. By our assumption that job $\tau_{q,\ell}$ with absolute deadline at t_d has a tardiness exceeding Δ, it follows that $\tau_{q,\ell}$ does not complete execution at or before $t_c = t_d + \Delta$.

Let τ_k^f and τ_k^m denote the sets of all fixed and migrating tasks, respectively, that are assigned to P_k. (Note that $\tau_k^m = \{\tau_i, \tau_j\}$.) Let *demand*$(\tau, t_0, t_c)$ denote the maximum time that jobs of tasks in τ could execute in the interval $[t_0, t_c)$ on Processor P_k (under the assumption that $\tau_{q,\ell}$ does not complete executing at t_c). We first determine *demand*(τ_k^m, t_0, t_c) and *demand*(τ_k^f, t_0, t_c).

By (16) and because migrating tasks have a higher priority than fixed tasks under EDF-fm, jobs of τ_i and τ_j that are released before t_0 and are assigned to P_k complete executing at or before t_0. Thus, every job of τ_i or τ_j that executes in $[t_0, t_c)$ on P_k is released in $[t_0, t_c)$. Also, every job released in $[t_0, t_c)$ and assigned to P_k places a demand for execution in $[t_0, t_c)$. The number of jobs of τ_i that are released in $[t_0, t_c)$ is at most $\left\lceil \frac{t_c - t_0}{p_i} \right\rceil$. By Lemma 2, at most $\left\lceil f_{i,k} \left\lceil \frac{t_c - t_0}{p_i} \right\rceil \right\rceil < f_{i,k}\left(\frac{t_c - t_0}{p_i} + 1\right) + 1$ of all the jobs of τ_i released in $[t_0, t_c)$ are assigned to P_k. Similarly, the number of jobs of τ_j that are assigned to P_k of all jobs of τ_i released in $[t_0, t_c)$ is less than $f_{j,k}\left(\frac{t_c - t_0}{p_j} + 1\right) + 1$. Each job of τ_i executes for at most e_i time units and that of τ_j for e_j time units. Therefore,

$$\begin{aligned} demand(\tau_k^m, t_0, t_c) &< \left(f_{i,k}\left(\frac{t_c - t_0}{p_i} + 1\right) + 1\right) \cdot e_i + \left(f_{j,k}\left(\frac{t_c - t_0}{p_j} + 1\right) + 1\right) \cdot e_j \\ &= s_{i,k}(t_c - t_0) + e_i(f_{i,k} + 1) + s_{j,k}(t_c - t_0) + e_j(f_{j,k} + 1) \qquad (17) \\ &\qquad \text{(by (6) and simplification).} \end{aligned}$$

By (14)–(16), and our assumption that the tardiness of $\tau_{q,\ell}$ exceeds Δ, any job of a fixed task that executes on P_k in $[t_0, t_c)$ is released at or after t_0 and has a deadline at or before t_d. The number of such jobs of a fixed task τ_f is at most $\left\lfloor \frac{t_d - t_0}{p_f} \right\rfloor$. Therefore,

$$\begin{aligned} demand(\tau_k^f, t_0, t_c) &< \sum_{\tau_f \in \tau_k^f} \left\lfloor \frac{t_d - t_0}{p_f} \right\rfloor \cdot e_f \\ &\leq (t_d - t_0) \sum_{\tau_f \in \tau_k^f} \frac{e_f}{p_f} \\ &\leq (t_d - t_0)(\rho_k - s_{i,k} - s_{j,k}) \qquad \text{(by (P3)).} \qquad (18) \end{aligned}$$

By (17) and (18), we have the following.

$$\begin{aligned} demand(\tau_k^f \cup \tau_k^m, t_0, t_c) &\leq s_{i,k}(t_c - t_0) + e_i(f_{i,k} + 1) + s_{j,k}(t_c - t_0) + e_j(f_{j,k} + 1) + (t_d - t_0)(\rho_k - s_{i,k} - s_{j,k}) \\ &= (s_{i,k} + s_{j,k})(t_c - t_0) + (s_{i,k} + s_{j,k})(t_0 - t_d) + e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) + (t_d - t_0)(\rho_k) \\ &= (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) + \rho_k(t_d - t_0) \end{aligned}$$

Because $\tau_{q,\ell}$ does not complete executing by time t_c, it follows that the total processor time available in the interval $[t_0, t_c) = t_c - t_0 < demand(\tau_k^f \cup \tau_k^m, t_0, t_c)$, *i.e.*,

$$\begin{aligned}
t_c - t_0 &< (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k}+1) + e_j(f_{j,k}+1) + \rho_k(t_d - t_0) \\
&= (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k}+1) + e_j(f_{j,k}+1) + (t_d - t_0) - (1-\rho_k)(t_d - t_0) \\
\Rightarrow\ t_c - t_d &< (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k}+1) + e_j(f_{j,k}+1) - (1-\rho_k)(t_d - t_0) \\
&\leq (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k}+1) + e_j(f_{j,k}+1) - p_q(1-\rho_k) \\
&\qquad \text{($\tau_{q,\ell}$ is released at or after t_0 and has a deadline at t_d, hence $t_d - t_0 \geq p_q$)} \\
\Rightarrow\ t_c - t_d &< \frac{e_i(f_{i,k}+1) + e_j(f_{j,k}+1) - p_q(1-\rho_k)}{1 - s_{i,k} - s_{j,k}} = \Delta. \qquad (19)
\end{aligned}$$

The above contradicts (15), and hence our assumption that the tardiness of $\tau_{q,\ell}$ exceeds Δ is incorrect. Therefore, (L) follows.

If only one migrating task τ_i is assigned to P_k, then e_j and $s_{j,k}$ are zero. Hence, a tardiness bound for any fixed task on P_k is given by

$$\frac{e_i(f_{i,k}+1) - p_q(1-\rho_k)}{1 - s_{i,k}}. \qquad (20)$$

If we let $m_{k,\ell}$, where $1 \leq \ell \leq 2$ denote the indices of the migrating tasks assigned to P_k, then by (L), a tardiness bound for EDF-fm is given by the following theorem. (If one or no migrating task is assigned to P_k, then $m_{k,2}$ and $m_{k,1}$ are to be taken to be zero, as are e_0, $f_{0,k}$, and $s_{0,k}$.)

Theorem 1 *On M processors, Algorithm* EDF-fm *ensures a tardiness of at most*

$$\frac{e_{m_{k,1}}(f_{m_{k,1},k}+1) + e_{m_{k,2}}(f_{m_{k,2},k}+1) - p_q(1-\rho_k)}{1 - s_{m_{k,1},k} - s_{m_{k,2},k}} \qquad (21)$$

for every task τ_q in τ where $U_{sum}(\tau) \leq \sum_{i=1}^{M} \rho_i$ and $u_{\max}(\tau) \leq \min(1/2, \min_{1\leq i\leq M} \rho_i)$, and τ_q is assigned to P_k.

Because (21) can be computed in constant time, the overall time complexity of computing a tardiness bound for τ is $\mathcal{O}(N)$. (21) increases as the execution costs and shares of the migrating tasks assigned to P_k increase, and could be high if the share of each migrating task is close to 1/2. However, because all tasks are light, in practice the sum of the shares of the migrating tasks assigned to a processor can be expected to be less than 1/2. Theorem 1 also suggests that the tardiness that results in practice could be reduced by choosing the set of migrating tasks carefully. Tardiness can also be reduced by distributing smaller pieces of work of migrating tasks than entire jobs. Some such techniques and heuristics are discussed in the next section.

6 Tardiness Reduction Techniques for EDF-fm

The problem of assigning tasks to processors such that the tardiness bound given by (21) is minimized is a combinatorial optimization problem with exponential time complexity. Hence, in this section, we propose methods and heuristics that can lower tardiness. We consider the technique of *period transformation* [34] as a way of distributing the execution of jobs of migrating tasks more evenly over their periods in order to reduce their adverse impact on fixed tasks. We also propose task assignment heuristics that can reduce the fraction of a processor's capacity consumed by migrating tasks. Finally, we show how to compute more accurate bounds than that given by (21) at the expense of more complex computations.

6.1 Job Slicing

The tardiness bound of EDF-fm given by Theorem 1 is in multiples of the execution costs of migrating tasks. This is a direct consequence of statically prioritizing migrating tasks over fixed tasks and the overload (in terms of the number of jobs) that a migrating task may place on a processor over short intervals. The deleterious effect of this approach on jobs of fixed tasks can be mitigated by "slicing" each job of a migrating task into *sub-jobs* that have lower execution costs, assigning appropriate deadlines to the sub-jobs, and distributing and scheduling sub-jobs in the place of whole jobs. For example, every job of a task with an execution cost of 4 time units and relative deadline of 10 time units can be sliced into two sub-jobs with execution cost and relative deadline of 2 and 5, respectively, per sub-job, or four sub-jobs with an execution cost of 1 and relative deadline of 2.5, per sub-job. Such a job-slicing approach, termed *period transformation*, was proposed by Sha and Goodman [34] in the context of RM scheduling on uniprocessors. Their purpose was to boost the priority of tasks that have larger periods, but are more important than some other tasks with shorter periods, and thus ensure that the more important tasks do not miss deadlines under overloads. However, with the job-slicing approach under EDF-fm, it may be necessary to migrate a job between its processors, and EDF-fm loses the property that a task that migrates does so only across job boundaries. Thus, this approach presents a trade-off between tardiness and migration overhead.

6.2 Task-Assignment Heuristics

Another way of lowering the actual tardiness observed in practice would be to lower the total share $s_{m_{k,1},k} + s_{m_{k,2},k}$ assigned to the migrating tasks on any processor P_k. In the task assignment algorithm ASSIGN-TASKS of Figure 2, if a low-utilization task is ordered between two high-utilization tasks, then it is possible that $s_{m_{k,1},k} + s_{m_{k,2},k}$ is arbitrarily close to one. For example, consider tasks τ_{i-1}, τ_i, and τ_{i+1} with utilizations $\frac{1-\epsilon}{2}$, 2ϵ, and $\frac{1-\epsilon}{2}$, respectively, and a task assignment wherein τ_{i-1} and τ_{i+1} are the migrating tasks of P_k with shares of $\frac{1-2\epsilon}{2}$ each, and τ_i is the only fixed task on P_k. Such an assignment, which can delay τ_i excessively if the periods of τ_{i-1} and τ_{i+1} are large, can be easily avoided by ordering tasks by (monotonically) decreasing utilization prior to the assignment phase. Note that with tasks ordered by decreasing utilization, of all the tasks not yet assigned to processors, the one with the highest utilization is always chosen as the next migrating task. Hence, we call this assignment scheme *highest utilization first*, or HUF. An alternative *lowest utilization first*, or LUF, scheme can be defined that assigns fixed tasks in the order of (monotonically) decreasing utilization, but chooses the task with the lowest utilization of all the unassigned tasks as the next migrating task. Such an assignment can be accomplished using the following procedure when a migrating task needs to be chosen: traverse the unassigned task array in reverse order starting from the task with the lowest utilization and choose the first task whose utilization is at least the capacity available in the current processor. In general, this scheme can be expected to lower the shares of migrating tasks. However, because the unassigned tasks have to be scanned each time a migrating task is chosen, the time complexity of this scheme increases to $\mathcal{O}(NM)$ (from $\mathcal{O}(N)$). This complexity can be reduced to $\mathcal{O}(N + M \log N)$ by adopting a binary-search strategy.

A third task-assignment heuristic, called *lowest execution-cost first*, or LEF, which is similar to LUF, can be defined by ordering tasks by execution costs, as opposed to utilizations. Fixed tasks are chosen in non-increasing order of execution costs; the unassigned task with the lowest execution cost, whose utilization is at least that of the available capacity in the current processor, is chosen as the next migrating task. The experiments reported in the next section show that LEF actually performs the best of these three task-assignment heuristics and that when combined with the job-slicing approach, can reduce tardiness dramatically in practice.

6.3 Including Non-Light Tasks

The primary reason for restricting all tasks to be light is to prevent the total utilization $u_i + u_j$ of the two migrating tasks τ_i and τ_j assigned to a processor from exceeding one. (As already noted, ensuring that migrating tasks do not miss their deadlines may not be possible otherwise.) However, if the number of non-light tasks is small in comparison to the number of light tasks, then it may be possible to avoid an undesirable assignment as described. In the simulation experiments discussed in Section 7, with no restrictions on per-task utilizations, the LUF approach could successfully assign approximately 78% of one million randomly-generated task sets on 4 processors. The success ratio dropped to approximately one-half when the number of processors increased to 16.

6.4 Processors with One Migrating Task

If the number of migrating tasks assigned to a processor P_k is one, then the commencement of the execution of a job $\tau_{i,j}$ of the only migrating task τ_i of P_k can be postponed to time $d(\tau_{i,j}) - e_i$, where $d(\tau_{i,j})$ is the absolute deadline of job $\tau_{i,j}$ (instead of beginning its execution immediately upon its arrival). This would reduce the maximum tardiness of the fixed tasks on P_k to $(e_i - p_q(1-\rho_k))/(1-s_{i,k})$ (from the value given by (20)). The reasoning is as follows. From the analysis in Section 5, tardiness of fixed tasks is bounded when the migrating task is not deferred, and hence, by the same analysis, is guaranteed to be bounded with deferred execution. This in turn implies that an arbitrary job of any fixed task completes execution. Taking t_c as the completion time of job $\tau_{q,\ell}$ when all the jobs execute for their worst-case execution times, where $\tau_{q,\ell}$ is as defined in Section 5, no job of τ_i with deadline later than t_c executes before t_c. (This is because, under deferred execution, each job of τ_i completes executing at its deadline. Hence, if $\tau_{q,\ell}$ completes execution at t_c, then neither is t_c the deadline of any job of τ_i nor does the first job with deadline after t_c commence execution by t_c.) Therefore, the number of jobs of τ_i released in the interval $[t_0, t_c)$ that can impact $\tau_{q,\ell}$ is at most $\left\lfloor \frac{t_c - t_0}{p_i} \right\rfloor \leq \frac{t_c - t_0}{p_i}$. This is one fewer job than that possible in the absence of deferred execution, which helps lower the tardiness bound derived by $\frac{e_i \cdot f_{i,k}}{1 - s_{i,k}}$. (Because tasks are independent and are preemptable, tardiness is guaranteed to not increase when one or more jobs execute for less than their worst-case execution times.) This technique is likely to be particularly effective on two-processor systems, where each processor would be assigned at most one migrating task only under EDF-fm, and on three-processor systems, where at most one processor would be assigned two migrating tasks.

6.5 Computing Less-Pessimistic Tardiness Bounds

Thus far in this section, we have discussed some techniques that can be used to lower the tardiness observed in practice and the bound computed using (21). We now describe how a more accurate tardiness bound can be computed for a given task assignment.

One major source of pessimism in the bound is the approximation of ceiling and floor operations during the analysis. This could be eliminated at the expense of more complex computations, wherein a bound is computed by iteratively computing a worst-case *response time*[5] for each task. The approach is similar to the time-demand [27] and the generalized time-demand analyses [25] used in conjunction with static-priority algorithms, and the response-time analysis developed for systems scheduled under EDF on uniprocessors [35].

Before continuing further, some definitions are in order. An interval $[t_1, t_2)$ is said to be *busy* for Processor P_k if the following hold. (In the description of a busy interval that follows, by P_k's tasks, we refer to both its fixed and migrating tasks, and by jobs of P_k's migrating tasks, we refer to jobs that are assigned to P_k.) **(i)** No job of any of P_k's tasks that is released before t_1 is pending at t_1; **(ii)** one or more jobs of P_k's tasks are released at t_1; and **(iii)** t_2 is the earliest time after t_1 such that no job released before t_2 is pending. Note that (i)–(iii) imply that P_k is continuously busy in $[t_1, t_2)$. A busy interval $[t_1, t_2)$ is said to be *in-phase for a fixed task* τ_i if a job of τ_i is released at t_1 and *in-phase for a migrating task* τ_i if a job of τ_i that begins a worst-case assignment sequence for P_k is released at t. By Lemma 2, at most $\lceil \ell \cdot f_{i,k} \rceil$ of any ℓ consecutive jobs of a migrating task τ_i are assigned to P_k. Therefore, $[t_1, t_2)$ is in-phase for τ_i if some job of τ_i is released at t_1, and if $\lceil \ell_t \cdot f_{i,k} \rceil$ of the jobs of τ_i are assigned to P_k in the interval $[t_1, t)$ for all $t_1 \leq t < t_2$, where ℓ_t denotes the number of jobs of τ_i released in that interval. A busy interval is said to be *tight* if all tasks release jobs as early as permissible after the release of their first jobs in the interval.

[5] The response time of a job is the difference between the time it completes execution and the time it is released.

In [35], the following has been shown for a task system scheduled under EDF on a uniprocessor: The largest response time of any job of a task τ_i (all tasks are fixed on a uniprocessor) released in a tight, busy interval that is in-phase for every task except perhaps τ_i is not lower than that of any job of τ_i released in any busy interval. Under EDF-fm, the same can be shown to hold for every fixed task. The reasoning is transformation-based and is as follows: By definition, no job is pending at the beginning of a busy interval; hence, transforming a busy interval that is not in-phase for a task τ_j (which is either fixed or migrating), by shifting left its jobs released in the interval, such that τ_j is in-phase, cannot decrease the demand due to τ_j in the busy interval that can compete with τ_i's jobs. In other words, since the interval is busy, the total competing work for τ_i's jobs cannot decrease. Similarly, the demand due to τ_j cannot decrease if its job releases are made tight, *i.e.*, if τ_j's jobs are released as early as permissible.

From generalized time-demand analysis for static-priority systems, we know that the worst-case response time for any job of τ_i occurs in a busy interval that is tight and is in-phase for τ_i and every higher-priority task. However, as implied by the discussion in the previous paragraph, under both EDF and EDF-fm, the worst case for τ_i need not be in a busy interval that is in-phase for τ_i. So, to compute a worst-case response time, and hence, a tardiness bound, for τ_i, all possible phasings of τ_i need to be considered. (Formally, τ_i is said to have a phase ϕ_i with respect to a busy interval $[t_1, t_2)$, where $0 \leq \phi_i < t_2 - t_1$, if the first job of τ_i in the interval is released at time $t_1 + \phi_i$.) Furthermore, for each phasing, the worst-case response times of all jobs of τ_i released in the busy interval when the jobs are released in a tight sequence need to be computed. The release time of every job of τ_i released in a tight, busy interval with phase $\phi_i = \phi + k \cdot p_i$ for τ_i, where $k \geq 1$ and $0 \leq \phi < p_i$, is the same as that of some job of τ_i released in an interval with phase ϕ. Hence, the worst-case response time of any job released in the second interval is at least that of some job released in the first interval, and it suffices to consider ϕ_i in the range $[0, p_i)$ only.

Based on the above discussion, we now give formulas for iteratively computing the tardiness bounds of fixed tasks. We first show how to determine the length of a longest possible busy interval.

Computing the longest busy interval length. A tight, busy interval that is in-phase for every task, including τ_i, is at least as long as any busy interval that is not in-phase for τ_i. Therefore, we will upper bound the lengths of busy intervals we are interested in by that of one that is tight and in-phase for all tasks. For brevity, we will refer to such an interval as simply a busy interval. Without loss of generality, we assume that the longest busy interval that we are considering starts at time zero. Letting B_k denote the length of a longest busy interval of P_k, B_k can be computed iteratively as follows. As in Section 5, τ_k^m and τ_k^f refer to the sets of fixed and migrating tasks, respectively, assigned to P_k. B_k^0 denotes the initial value of B_k and is given by the following (since each task has a job released at the start of the interval that needs to complete execution).

$$B_k^0 = \sum_{\tau_h \in \tau_k^m} e_h + \sum_{\tau_h \in \tau_k^f} e_h \tag{22}$$

If B_k^i, where $i \geq 0$, denotes the value of B_k in the i^{th} iteration, then P_k is continuously busy at least until B_k^i. The length of the busy interval could be longer if not all jobs that can potentially be released before B_k^i complete executing by B_k^i. Therefore, B_k^{i+1}, the value of B_k in the $(i+1)^{\text{st}}$ iteration, is given by the execution costs of all the jobs that can be released in an interval of length B_k^i, and hence, is

$$B_k^{i+1} \;=\; \sum_{\tau_h \in \tau_k^m} \left\lceil \left\lceil \frac{B_k^i}{p_h} \right\rceil \cdot f_{h,k} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau_k^f} \left\lceil \frac{B_k^i}{p_h} \right\rceil \cdot e_h. \tag{23}$$

The iterations terminate when $B_k^i = B_k^{i+1}$ for some $i \geq 0$, *i.e.*, B_k is given by the following.

$$B_k \;=\; \min_{i \geq 0} \{B_k^i \mid B_k^i = B_k^{i+1}\}$$

We next show that termination is guaranteed. For simplicity, we assume that the execution costs and periods of all tasks are integers. Let $s_{h,k} = \frac{x_{h,k}}{y_{h,k}}$, where $x_{h,k}$ and $y_{h,k}$ are positive integers that are relatively prime. Let Δ denote the least common multiple (lcm) of the periods of all fixed and migrating tasks and the product $e_h \cdot y_{h,k}$ for each migrating task τ_h. Then, for all $B_k^i \leq \Delta$, by (23) and (6), $B_k^{i+1} \leq \sum_{\tau_h \in \tau_k^m} \left\lceil \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot \frac{s_{h,k} \cdot p_h}{e_h} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau_k^f} \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot e_h = \sum_{\tau_h \in \tau_k^m} \left\lceil \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot \frac{x_{h,k} \cdot p_h}{y_{h,k} \cdot e_h} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau_k^f} \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot e_h$. Since Δ is as defined, p_h for each fixed and migrating task, and $e_h \cdot y_{h,k}$ for each migrating task divide Δ evenly, and hence, $B_k^{i+1} \leq \sum_{\tau_h \in \tau_k^m} \frac{\Delta}{p_h} \cdot \frac{x_{h,k} \cdot p_h}{y_{h,k} \cdot e_h} \cdot e_h + \sum_{\tau_h \in \tau_k^f} \frac{\Delta}{p_h} \cdot e_h = \sum_{\tau_h \in \tau_k^m} \Delta \cdot s_{h,k} + \sum_{\tau_h \in \tau_k^f} \frac{\Delta}{p_h} \cdot e_h$. Because the sum of the shares of the migrating tasks and the utilizations of the fixed tasks assigned to each processor is at most one, the right-hand side of the above inequality is at most Δ. Thus, the computation converges at least when $B_k^i = \Delta$, and hence, B_k is at most Δ.

Computing tardiness bounds. We first describe how to compute worst-case completion times, and hence, worst-case response times, for jobs of a fixed task τ_q released within a tight, busy interval, with a phase or offset of ϕ_q for τ_q. (A tardiness bound for τ_q may then be determined from the job completion times computed. Again, without loss of generality, we assume that each busy interval considered starts at time zero. Therefore, worst-case completion times directly yield worst-case response times.) The number of jobs of τ_q, denoted J, released in such a busy interval is at most $\left\lceil \frac{B_k - \phi_q}{p_q} \right\rceil$, and the deadline of the J^{th} job is at or after B_k. Since B_k is the length of a longest busy interval, and hence, an upper bound on the length of the interval under consideration, one of the following holds: **(i)** the J^{th} job completes executing by B_k, *i.e.*, at or before its deadline, and hence, its tardiness is zero; **(ii)** the processor is idle at some time t before B_k and the J^{th} job is released after t. If (i) holds, then since the tardiness of the J^{th} job is zero, its response time need not be computed, and if (ii) holds, then the J^{th} job is not released in the busy interval under consideration. Therefore, in either case, in order to determine a tardiness bound for τ_q, it suffices to determine the worst-case response times of only the first $J - 1 = \left\lceil \frac{B_k - \phi_q}{p_q} \right\rceil - 1$ jobs released in the interval. Without loss of generality, we denote the ℓ^{th} job of τ_q released in a busy interval as $\tau_{q,\ell}$.

It is quite possible that the tight, busy interval with phase ϕ_q ends before $\tau_{q,\ell}$'s release, $r_{q,\ell}$, or fixed jobs with deadlines after that of $\tau_{q,\ell}$, $d_{q,\ell}$, execute before $r_{q,\ell}$. In the former case, clearly $\tau_{q,\ell}$ and later jobs are not part of the busy interval. In the latter case, the interval is not really busy with respect to $\tau_{q,\ell}$ and later jobs, and response times for these jobs will be bounded by those of other jobs released possibly with a different phase. Hence, in either case, computation of response times can be omitted for these jobs based on whether the total demand up to $r_{q,\ell}$ due to migrating tasks and fixed jobs with deadlines at most $d_{q,\ell}$ is less than r_{q_ℓ}. (For such jobs, the formulas below may in fact yield lower than the actual response times. However, this is not an issue since we are concerned with determining the maximum possible response time only.)

Let C_{q,ℓ,ϕ_q}, where $1 \leq \ell \leq J-1 = \left\lceil \frac{B_k - \phi_q}{p_q} \right\rceil - 1$, denote the worst-case completion time, relative to the beginning of the busy interval (which is time zero by our assumption), of $\tau_{q,\ell}$ when τ_q's phase is ϕ_q. Then, C_{q,ℓ,ϕ_q} can be computed iteratively as follows. Let C^0_{q,ℓ,ϕ_q} denote the intial value. Since $\tau_{q,\ell}$ is released at time $(\ell - 1) \cdot p_q + \phi_q$, and the longest busy interval ends at time B_k, an initial estimate is given by

$$C^0_{q,\ell,\phi_q} = \min(B_k - e_q, (\ell - 1) \cdot p_q + \phi_q) + e_q. \tag{24}$$

All jobs of migrating tasks released before C^0_{q,ℓ,ϕ_q} and assigned to P_k contend for execution before C^0_{q,ℓ,ϕ_q}. The deadline of $\tau_{q,\ell}$ is given by $d(\tau_{q,\ell}) = \ell \cdot p_q + \phi_q$. Hence, jobs of fixed tasks with deadlines at most $d(\tau_{q,\ell})$ and released before C^0_{q,ℓ,ϕ_q} also contend for execution before C^0_{q,ℓ,ϕ_q}. Hence, C_{q,ℓ,ϕ_q} can be revised iteratively as follows for $i \geq 0$.

$$C^{i+1}_{q,\ell,\phi_q} \quad = \quad \sum_{\tau_h \in \tau^m_k} \left\lceil \left\lceil \frac{C^i_{q,\ell,\phi_q}}{p_h} \right\rceil \cdot f_{h,k} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau^f_k} \min\left(\left\lceil \frac{C^i_{q,\ell,\phi_q}}{p_h} \right\rceil, \left\lfloor \frac{\ell \cdot p_q + \phi_q}{p_h} \right\rfloor \right) \cdot e_h \tag{25}$$

The iterations can be terminated when convergence is reached, *i.e.*, when $C^{i+1}_{q,\ell,\phi_q} = C^i_{q,\ell,\phi_q}$ for some $i \geq 0$. Convergence is guaranteed because C_{q,ℓ,ϕ_q} is at most B_k for all ℓ, ϕ_q.

Earlier we explained that it suffices to consider ϕ_q in the range $[0, p_q)$. This range can be lowered further in some cases by noting that if $\phi_q \geq B_k - p_q$, then the number of jobs, $\left\lceil \frac{B_k - \phi_q}{p_q} \right\rceil - 1$, of τ_q whose worst-case response times need to be computed is at most zero. Therefore, it suffices to consider ϕ_q in the range $[0, \min(p_q, B_k - p_q))$ only, and assuming that all task parameters are integral, C_{q,ℓ,ϕ_q} needs to be computed for all integers ϕ_q in $[0, \min(p_q - 1, B_k - p_q - 1)]$. Since $d(\tau_{q,\ell}) = \ell \cdot p_q + \phi_q$, a tardiness bound for τ_q is given by the following.

$$\mathit{tardiness}(\tau_q) \leq \max_{0 \leq \phi_q \leq \min(p_q - 1, B_k - p_q - 1)} \left\{ \max_{1 \leq \ell \leq (\left\lceil \frac{B_k - \phi_q}{p_q} \right\rceil - 1)} \{\max(C_{q,\ell,\phi_q} - \ell \cdot p_q - \phi_q, 0)\} \right\}$$

Though convergence is guaranteed while computing the length of the busy interval and worst-case response times, the length of the busy interval, and hence, the number of iterations, could be exponential in N. Similarly, the number of jobs whose response times have to be computed could be exponential.

Numerical example. Let us consider computing tardiness bounds of fixed tasks assigned to P_1 in the task system in Example 1. In this example, $\tau_1^f = \{\tau_1(5,20), \tau_2(3,10)\}$ and $\tau_1^m = \{\tau_3(1,2)\}$. Further, $s_{3,1} = \frac{9}{20}$, hence, $x_{3,1} = 9$ and $y_{3,1} = 20$. Therefore, $e_3 \cdot y_{3,1} = 1 \cdot 20 = 20$. $\text{lcm}(p_1, p_2, p_3, e_3 \cdot y_{3,1}) = \text{lcm}(20, 10, 2, 20) = 20$. Also, $f_{3,1} = s_{3,1} \cdot \frac{p_3}{e_3} = \frac{9}{20} \cdot \frac{2}{1} = \frac{9}{10}$.

We will first compute the length, B_1, of a longest possible busy interval, on P_1. Using (22), $B_1^0 = e_3 + e_1 + e_2 = 9$. By (23),

$$
\begin{aligned}
B_1^1 &= \left\lceil \left\lceil \frac{B_1^0}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \left\lceil \frac{B_1^0}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{B_1^0}{p_2} \right\rceil \cdot e_2 \\
&= \left\lceil \left\lceil \frac{9}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \left\lceil \frac{9}{20} \right\rceil \cdot 5 + \left\lceil \frac{9}{10} \right\rceil \cdot 3 \\
&= 5 + 5 + 3 = 13, \\
B_1^2 &= \left\lceil \left\lceil \frac{B_1^1}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \left\lceil \frac{B_1^1}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{B_1^1}{p_2} \right\rceil \cdot e_2 \\
&= \left\lceil \left\lceil \frac{13}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \left\lceil \frac{13}{20} \right\rceil \cdot 5 + \left\lceil \frac{13}{10} \right\rceil \cdot 3 \\
&= 7 + 5 + 6 = 18, \\
B_1^3 &= \left\lceil \left\lceil \frac{B_1^2}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \left\lceil \frac{B_1^2}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{B_1^2}{p_2} \right\rceil \cdot e_2 \\
&= \left\lceil \left\lceil \frac{18}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \left\lceil \frac{18}{20} \right\rceil \cdot 5 + \left\lceil \frac{18}{10} \right\rceil \cdot 3 \\
&= 9 + 5 + 6 = 20, \\
B_1^4 &= \left\lceil \left\lceil \frac{B_1^3}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \left\lceil \frac{B_1^3}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{B_1^3}{p_2} \right\rceil \cdot e_2 \\
&= \left\lceil \left\lceil \frac{20}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \left\lceil \frac{20}{20} \right\rceil \cdot 5 + \left\lceil \frac{20}{10} \right\rceil \cdot 3 \\
&= 9 + 5 + 6 = 20.
\end{aligned}
$$

Since $B_1^4 = B_1^3$, the procedure terminates with the computation of B_1^4, and $B_1 = 20$.

We now compute tardiness bounds for fixed tasks of P_1 by computing their worst-case response times. We begin with τ_1. As explained earlier, it suffices to consider ϕ_q in the range $[0, \min(p_q - 1, B_k - p_q - 1)]$ for each fixed task τ_q on P_k. Since $B_1 = 20$, the range for ϕ_1 is $[0, -1]$, which is empty. This implies that tardiness for τ_1 is zero.

We next consider τ_2, for which, ϕ_2 is in the range $[0, 9]$. The number of jobs of τ_2 for which worst-case response times need to be determined is given by $J - 1 = \left\lceil \frac{B_1 - \phi_2}{p_2} \right\rceil - 1 = \left\lceil \frac{20 - \phi_2}{10} \right\rceil - 1 = 1$, for all $0 \le \phi_2 \le 9$. We compute the response time for the first job (*i.e.*, $\ell = 1$) of τ_2, which is $\tau_{2,1}$, when phase $\phi_2 = 0$. By (24), $C_{2,1,0}^0 = \min(20 - 3, 0) + 3 = 3$. By (25),

$$
\begin{aligned}
C_{2,1,0}^1 &= \left\lceil \left\lceil \frac{C_{2,1,0}^0}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \min\left(\left\lceil \frac{C_{2,1,0}^0}{p_1} \right\rceil, \left\lfloor \frac{\ell \cdot p_2 + \phi_2}{p_1} \right\rfloor \right) \cdot e_1 + \min\left(\left\lceil \frac{C_{2,1,0}^0}{p_2} \right\rceil, \left\lfloor \frac{\ell \cdot p_2 + \phi_2}{p_2} \right\rfloor \right) \cdot e_2 \\
&= \left\lceil \left\lceil \frac{3}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \min\left(\left\lceil \frac{3}{20} \right\rceil, \left\lfloor \frac{1 \cdot 10 + 0}{20} \right\rfloor \right) \cdot 5 + \min\left(\left\lceil \frac{3}{10} \right\rceil, \left\lfloor \frac{1 \cdot 10 + 0}{10} \right\rfloor \right) \cdot 3
\end{aligned}
$$

$$
\begin{aligned}
&= 2 + 0 + 3 = 5, \\
C^2_{2,1,0} &= \left\lceil \left\lceil \frac{C^1_{2,1,0}}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \min\left(\left\lceil \frac{C^1_{2,1,0}}{p_1} \right\rceil, \left\lfloor \frac{\ell \cdot p_2 + \phi_2}{p_1} \right\rfloor \right) \cdot e_1 + \min\left(\left\lceil \frac{C^1_{2,1,0}}{p_2} \right\rceil, \left\lfloor \frac{\ell \cdot p_2 + \phi_2}{p_2} \right\rfloor \right) \cdot e_2, \\
&= \left\lceil \left\lceil \frac{5}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \min\left(\left\lceil \frac{5}{20} \right\rceil, \left\lfloor \frac{1 \cdot 10 + 0}{20} \right\rfloor \right) \cdot 5 + \min\left(\left\lceil \frac{5}{10} \right\rceil, \left\lfloor \frac{1 \cdot 10 + 0}{10} \right\rfloor \right) \cdot 3 \\
&= 3 + 0 + 3 = 6, \\
C^3_{2,1,0} &= \left\lceil \left\lceil \frac{C^0_{1,1,2}}{p_3} \right\rceil \cdot f_{3,1} \right\rceil \cdot e_3 + \min\left(\left\lceil \frac{C^0_{1,1,2}}{p_1} \right\rceil, \left\lfloor \frac{\ell \cdot p_2 + \phi_2}{p_1} \right\rfloor \right) \cdot e_1 + \min\left(\left\lceil \frac{C^0_{1,1,2}}{p_2} \right\rceil, \left\lfloor \frac{\ell \cdot p_2 + \phi_2}{p_2} \right\rfloor \right) \cdot e_2 \\
&= \left\lceil \left\lceil \frac{6}{2} \right\rceil \cdot \frac{9}{10} \right\rceil \cdot 1 + \min\left(\left\lceil \frac{6}{20} \right\rceil, \left\lfloor \frac{1 \cdot 10 + 0}{20} \right\rfloor \right) \cdot 5 + \min\left(\left\lceil \frac{6}{10} \right\rceil, \left\lfloor \frac{1 \cdot 10 + 0}{10} \right\rfloor \right) \cdot 3 \\
&= 3 + 0 + 3 = 6.
\end{aligned}
$$

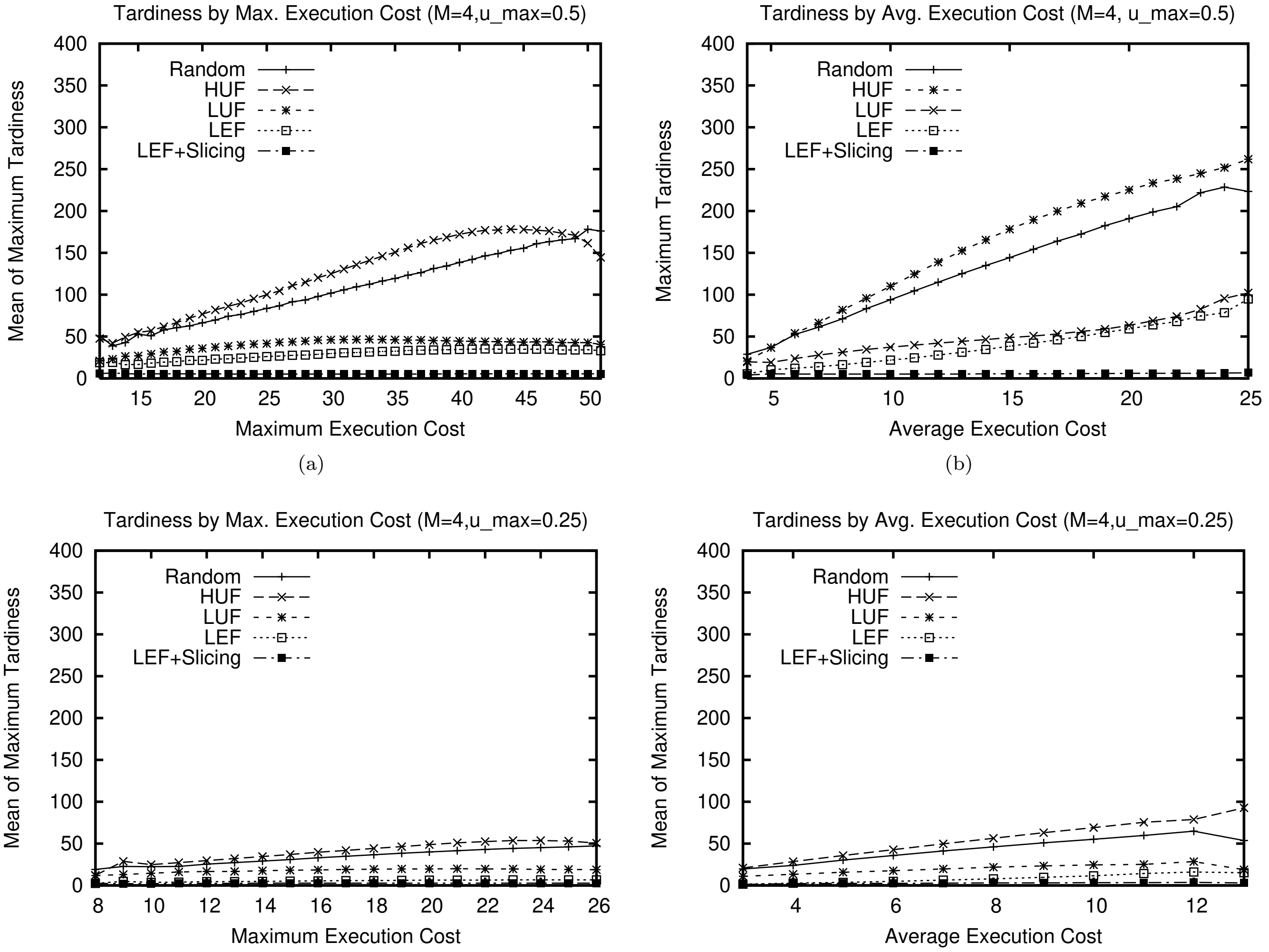
Thus, convergence is reached after four iterations and the worst-case response time of $\tau_{2,1} = 6$. Since $d(\tau_{2,1}) = 10$, $\tau_{2,1}$'s tardiness is zero. It can similarly be verified that tardiness for $\tau_{2,1}$ is zero for every ϕ_2 in $[1, 9]$. Hence, tardiness bounds for both τ_1 and τ_2 are zero. On the other hand, tardiness bounds computed for τ_1 and τ_2 using the formula in Section 6.4 are 9.01 and 5.45, respectively.

7 Simulation-Based Evaluation

In this section, we describe the results of four sets of simulation experiments conducted using randomly-generated task sets to evaluate EDF-fm and the heuristics described in Section 6.

The experiments in the first set evaluate the various task assignment heuristics for $M = 4$ and $M = 8$ (where M is the number of processors), and $u_{\max} = 0.25$ and $u_{\max} = 0.5$ (where $u_{\max}$ is the maximum utilization of any task in a task set). For each M and $u_{\max}$, 10^6 task sets were generated. Each task set τ was generated as follows: New tasks were added to τ as long as the total utilization of τ was less than M. For each new task τ_i, first, its period p_i was generated as a uniform random number in the range $[1.0, 100.0]$; then, its execution cost was chosen randomly in the range $[u_{max}, u_{\max} \cdot p_i]$. The last task was generated such that the total utilization of τ exactly equaled M. The generated task sets were classified by maximum and average execution costs (denoted $e_{\max}$ and e_{avg}). The tardiness bound given by (21) was computed for each task set under a random task assignment and also under heuristics HUF, LUF, and LEF. The average value of the tardiness bound for task sets in each group under each classification and heuristic was then computed. The results for the groups classified by $e_{\max}$ and e_{avg} for $M = 4$ and $u_{\max} = 0.5$ are shown in insets (a) and (b), respectively, of Figure 10. Insets (c) and (d) contain the results under the same classifications for the same M but for $u_{\max} = 0.25$. Results for $M = 8$ are shown in Figure 11. (99% confidence intervals were also computed but are omitted due to scale.)

Results for task sets grouped by $u_{\max}$ and u_{avg} are shown in Figure 12 for $M = 4$ and $M = 8$. Data for these results also come from 10^6 task sets. However, for this subset of experiments, tasks were generated by uniformly choosing an execution cost in the range $[1.0, 20.0]$ and a utilization in the range $[u_{\min}, u_{\max}]$; the pair $(u_{\min}, u_{\max})$



(a) (b) (c) (d)

Figure 10: Tardiness bounds under different task assignment heuristics for $M = 4$ and $u_{\max} = 0.5$ by **(a)** $e_{\max}$ and **(b)** e_{avg}, and for $M = 4$ and $u_{\max} = 0.25$ by **(c)** $e_{\max}$ and **(d)** e_{avg}.

for each task set was uniformly chosen from those in the set $\{(0.0, 0.2), (0.0, 0.4), (0.1, 0.5), (0.3, 0.5)\}$. This strategy was used so that a sufficient number of task sets fall under each u_{avg} group.

From the plots, we first observe that there is only a slight increase in the tardiness bounds as the number of processors is increased from four to eight. This is because the tardiness bound given by (21) is independent of M. However, the maximum of the tardiness bounds computed for all the tasks can be expected to increase as the number of processors, and hence, the number of tasks increase. The increase, however, seems to be negligible.

Coming to the comparison of the different heuristics, the plots show that LEF guarantees the minimum tardiness of the four task-assignment approaches. LUF is the next best with the difference between LEF and LUF being wider on $M = 8$ processors than on $M = 4$ processors. Another interesting observation is that HUF performs worse than even Random most of the time. Under LEF, tardiness is quite low (approximately 8 time units mostly) for $u_{\max} = 0.25$ (insets (c) and (d) of Figures 10 and 11 and insets (a) and (c) of Figure 12), which suggests that

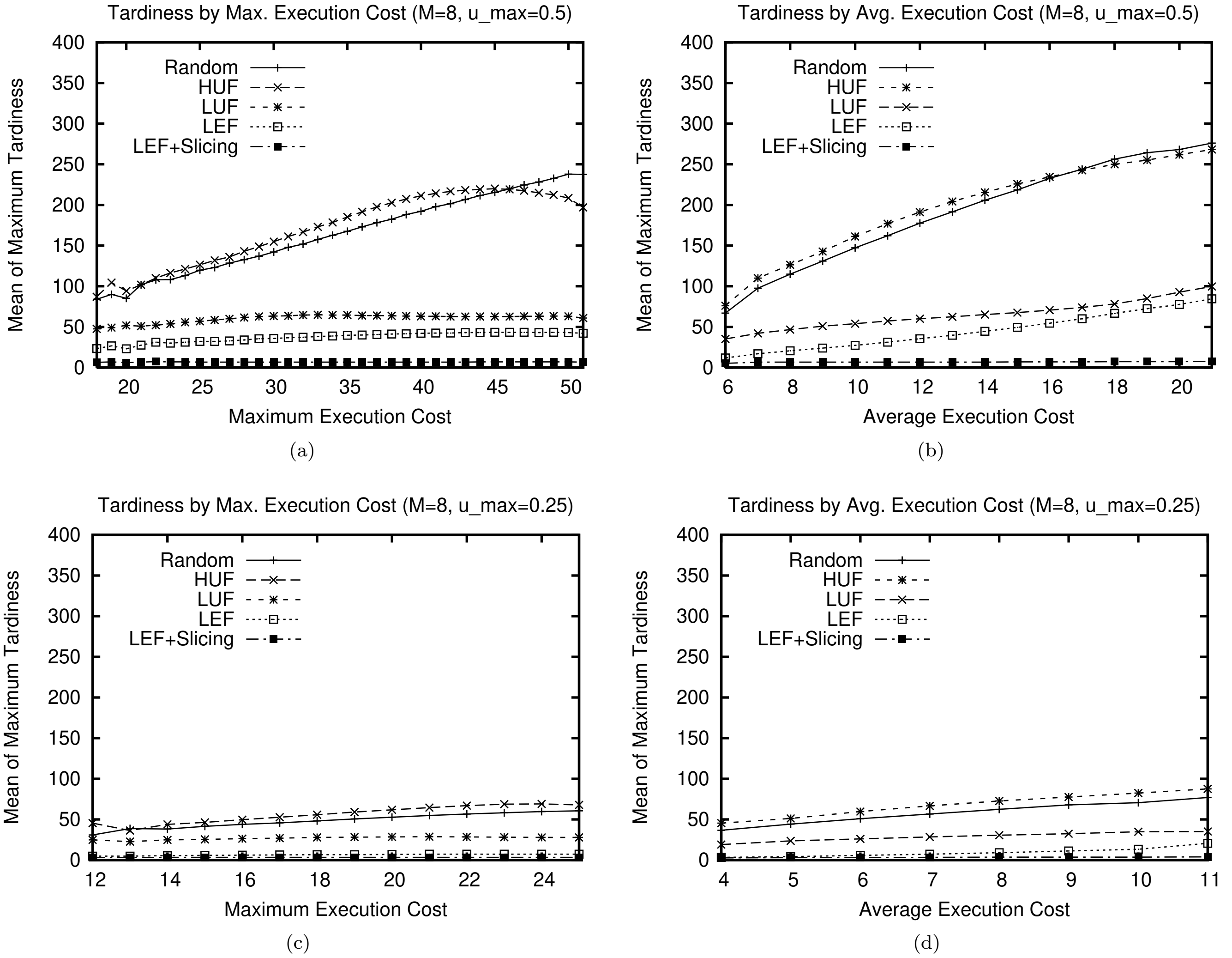
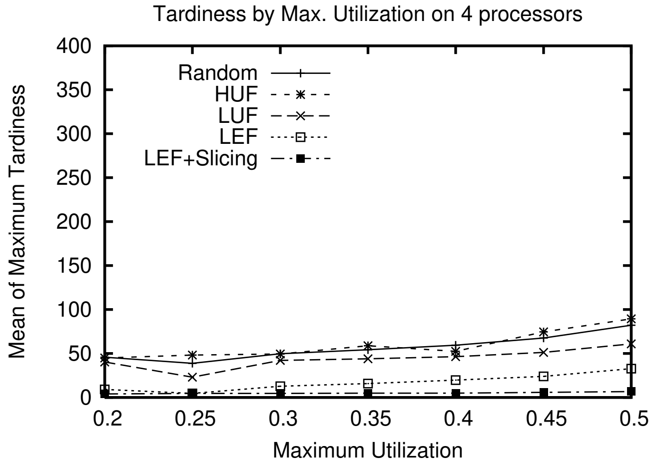
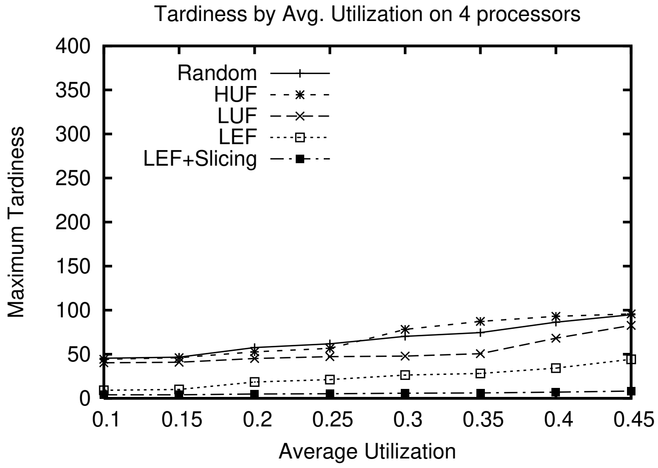


Figure 11: Tardiness bounds under different task assignment heuristics for $M = 8$ and $u_{\max} = 0.5$ by **(a)** $e_{\max}$ and **(b)** e_{avg}, and for $M = 8$ and $u_{\max} = 0.25$ by **(c)** $e_{\max}$ and **(d)** e_{avg}.

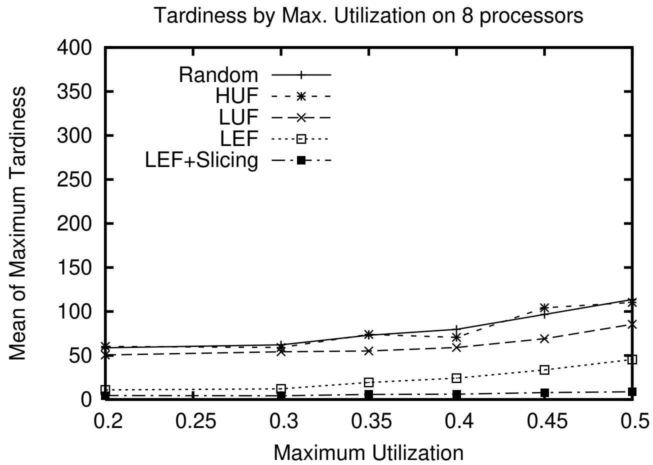
LEF may be a reasonable strategy for such task systems. Tardiness increases with increasing $u_{\max}$, but is still a reasonable value of 25 time units only for $e_{\text{avg}} \leq 10$ when $u_{\max} = 0.5$. However, for $e_{\text{avg}} = 20$, tardiness exceeds 75 time units when $M = 8$, which may not be acceptable. For such systems, tardiness can be reduced by using the job-slicing approach, at the cost of increased migration overhead. Therefore, in an attempt to determine the reduction possible with the job-slicing approach, we also computed the tardiness bound under LEF assuming that each job of a migrating task is sliced into sub-jobs with execution costs in the range $[1, 2)$. This bound is also plotted in the figures referred to above. For $u_{\max} = 0.5$, we found the bound to settle to approximately 7–8 time units, regardless of the execution costs and individual task utilizations. (When $u_{\max} = 0.25$, tardiness is only 1–2 time units under LEF with job slicing.) In our experiments, on average, a seven-fold decrease in tardiness was observed with job slicing with a granularity of one to two time units per sub-job. However, a commensurate increase in the number of migrations is also inevitable.



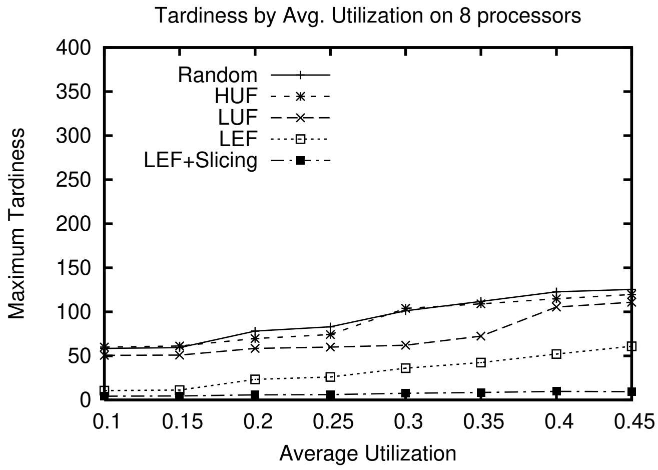
(a)



(b)



(c)

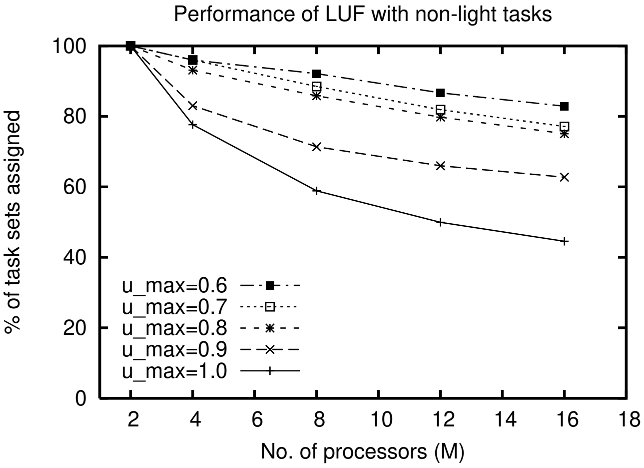


(d)

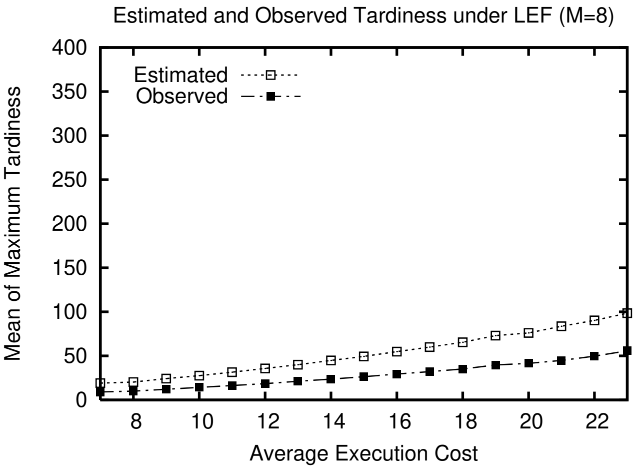
Figure 12: Tardiness bounds under different task assignment heuristics for **(a)** $M = 4$ and $u_{\max}$ **(b)** $M = 4$ and u_{avg} **(c)** $M = 8$ and $u_{\max}$ **(d)** $M = 8$ and u_{avg}. In all the graphs, $e_{\max} = 20$ and $e_{\text{avg}} = 10$.

Overall, the results indicate that the tardiness bounds guaranteed may be tolerable if task execution costs are not high and the LEF strategy is used for task assignment.

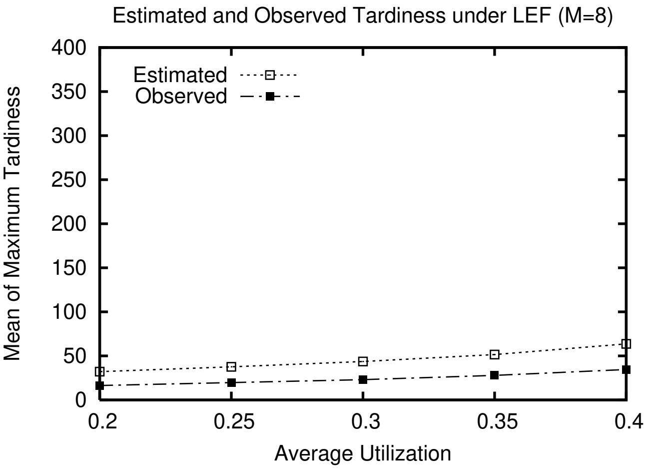
The second set of experiments evaluates the different heuristics in their ability to successfully assign task sets that contain non-light tasks also. Task sets were generated using the same procedure as that described for the first set of experiments above, except that $u_{\max}$ was varied between 0.6 and 1.0 in steps of 0.1. All of the four approaches could assign 100% of the task sets generated for $M = 2$, as expected. For higher values of M, the success ratio plummeted for all but the LUF approach. The percentage of task sets that LUF could successfully assign for varying M and $u_{\max}$ is shown in Figure 13(a). LEF performed next best (graphs not provided). However, even when $u_{\max} = 0.6$, its success percentage is approximately 79% when $M = 4$ and 24% when $M = 16$; the corresponding values are approximately 23.9% and 0.3%, respectively, when $u_{\max}$ is increased to 1.0. In this set of experiments also, HUF almost always performed worse than Random, and its success percentage was close to zero



(a)



(b)



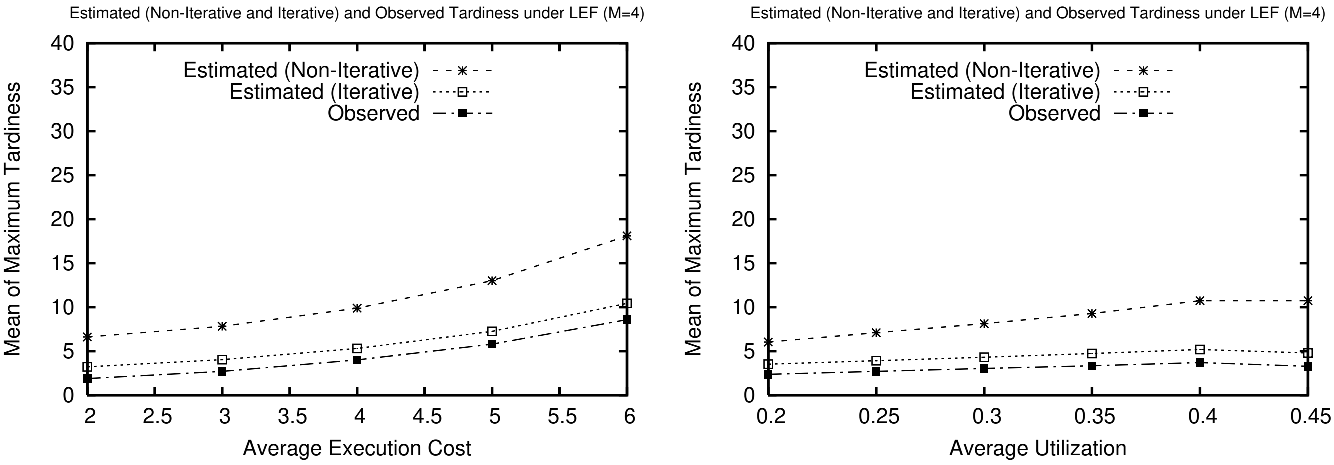
(c)

Figure 13: **(a)**Percentage of randomly-generated task sets with non-light tasks successfully assigned by the LUF heuristic. **(b) & (c)** Comparison of estimated and observed tardiness under EDF-fm-LEF by **(b)** average execution cost and **(c)** average utilization.

except when $M = 4$.

The third set of experiments was designed to evaluate the pessimism in the tardiness bound of (21). 300,000 task sets were generated with $u_{\max} = 0.5$ and $U_{sum} = 8$. The tardiness bound estimated by (21) under the LEF task assignment heuristic was computed for each task set. A schedule under EDF-fm-LEF for 100,000 time units was also generated for each task set (when each task releases jobs in a synchronous, periodic manner) and the actual maximum tardiness observed was noted. (The time limit of 100,000 was determined by trial-and-error as an upper bound on the time within which tardiness converged for the tasks sets generated.) Plots of the average of the estimated and observed values for tasks grouped by e_{avg} and u_{avg} are shown in insets (b) and (c) of Figure 13, respectively. In general, we found that actual tardiness is only approximately half of the estimated value.

Finally, experiments were run to compare the bounds computed iteratively, which could require exponential



(a) (b)

Figure 14: Comparison of tardiness estimated by the iterative formulas to that estimated by the closed-form formula in (21) and observed tardiness under LEF task assignment by **(a)** average execution cost and **(b)** average utilization.

time, to the actual tardiness. For this set of experiments, to facilitate computations, both periods and execution costs were chosen to be integers. Further, to reasonably constrain the length of the busy interval, the maximum period was restricted to 20, and some odd values such as 13, 17, and 19 were forbidden. Results are shown in Figure 14 and indicate that the bounds computed using this approach are very close to actual tardiness observed.

8 Related Work

A primary goal of research on soft real-time systems is to provide weaker guarantees on meeting timing constraints, where permissible, in an attempt to improve resource utilization. Most prior work on soft real-time scheduling has focussed on uniprocessor systems only. On uniprocessors, several real-time models that differ in the type of soft real-time guarantees they provide have been proposed in the literature. We will classify the models as either *deterministic* or *probabilistic* based on the predominant nature of the soft real-time guarantees they provide.

Deterministic models. Among the early deterministic models is the *skippable periodic task model* of Koren and Shasha [23]. In this model, jobs of a task may be *skipped*, *i.e.*, may either miss their deadlines or be aborted at any time, as long as there is a minimum separation between two consecutive skips. Koren and Shasha showed that utilizing skips optimally is NP-hard and proposed algorithms that can exploit skips. Variations on this theme have been proposed by others either independently or as follow-up work. In [20], Hamdaoui and Ramanathan introduced the (m, k)-*firm deadline model*, in which at least m jobs in every k consecutive jobs should meet their deadlines, and devised algorithms that can provide probabilistic guarantees on meeting the (m, k) constraint. The *window-constrained model* [40] is another similar model, whereas the *weakly-hard real-time model* [12] strives to provide a general framework for the specification of soft real-time constraints. In particular, in the weakly-hard model, a specification for soft real-time constraints (referred to as *weakly-hard constraints* by the authors), which can allow multiple constraints to be associated with each task, and an algebra that relates different constraints, are developed. Schedulability analysis for such task systems under fixed-priority scheduling is also presented. The weakly-hard real-time model also allows jobs that do not meet their deadlines to complete late, but no bound on tardiness is provided.

The models discussed above allow some jobs to be treated as optional and their executions to either complete late or be discarded entirely. In contrast, the imprecise computation model [29] is another deterministic model in which some portion of every job can be optional and can be discarded. In this model, each job is composed of a mandatory part and an optional part. The goal is to execute the mandatory part of each job by the job deadline, and to schedule the optional part so that one or more performance metrics, such as, the number of jobs whose optional parts are discarded, the total amount of time by which all the optional parts are late, etc., are optimized. In [7], Aydin *et al.* associate a reward function, which is non-decreasing with the amount of work completed, with the optional parts of the jobs of each periodic task, and show how to schedule imprecise tasks such that the weighted average reward is maximized.

The soft real-time model considered in this paper also provides deterministic guarantees. This model, where sporadic tasks with implicit deadlines have a tardiness threshold, has not been considered much in the context of uniprocessors. This is due to the fact that for systems that are *not* overloaded, scheduling under EDF is sufficient to ensure that each job completes execution by its deadline, and there is no scope of allowing a higher utilization even if bounded tardiness is tolerable. This model can alternatively be viewed as one in which timing constraints are hard, but the relative deadlines of the sporadic tasks are *larger* than their periods.[6] When viewed in this alternative manner, the model has been the focus of some research in the context of *deadline-monotonic* scheduling on uniprocessors. In [25], Lehoczky provides a demand-based test, which may require exponential time, to determine the schedulability of task systems that use the model, and also derives utilization bounds when the relative deadline of each task is a multiple (greater than one) of its period. However, determining the maximum time after the end of its period that any job may complete executing has not been addressed.

Probabilistic models. For some tasks, such as those in video-conferencing applications or animation games, the inter-arrival time between two consecutive jobs or the execution requirements of different jobs can vary widely over time. For such systems, reserving resources based on worst-case parameters (shortest inter-arrival time and longest execution time) can be extremely wasteful. One way of improving resource utilization for such systems is to model task parameters probabilistically (using probability distributions for inter-arrival times or execution times)

[6]It should be noted that if the relative deadlines of all tasks are not increased by equal amounts, then it is possible for a task with a shorter period to have a larger relative deadline, or vice versa. Hence, when viewed in this alternative manner, task priorities determined based on relative deadlines or job priorities based on absolute deadlines may not match those determined in our model in which relative deadlines equal periods.

and provide probabilistic guarantees on meeting deadlines. Examples of research in this direction can be found in [38, 6, 1, 2, 17]. Within the probabilistic domain, Lehoczky's *real-time queueing theory* [26] strives to provide a framework for combining the timing constraints of real-time scheduling with the stochastic elements of queueing theory. This theory may be used to determine the fraction of jobs missing their deadlines in either single-queue (*i.e.*, single task), single-server systems [18], or acyclic networks of servers with multiple queues [24] under heavy traffic conditions.[7]

Almost all of the work on the models described above is concerned with dealing with overload on uniprocessors, *i.e.*, scenarios in which the total system utilization exceeds the available capacity, which is 1.0. (In general, deterministic models are used when the overload is over longer terms, such as system lifetimes. Probabilistic models are more suited for systems that may be overloaded occasionally for short durations but whose average loads do not exceed the available capacity. Combinations of both are also possible.) However, in the context of multiprocessors, there exist scheduling algorithms under which guarantees on timeliness that can be provided are not known when the total utilization exceeds the schedulable utilization of the algorithm (but not the available processing capacity). Since the worst-case schedulable utilization of the concerned algorithms is $(M+1)/2$, possible guarantees on timeliness are not known even when the system is much below full load. This is true of most of the known algorithms except optimal Pfair algorithms, necessitating a study of the model of this paper. We conjecture that on multiprocessors, results based on this model may, in fact, be necessary in dealing with overload.

On multiprocessors, soft Pfair-based real-time scheduling has previously been considered in [37] and [15], where tardiness bounds are derived for a suboptimal Pfair scheduling algorithm that is less expensive than optimal algorithms. As already mentioned, deriving tardiness bounds under global preemptive EDF has been considered in [16] and [39], and that under non-preemptive EDF in [16].

9 Concluding Remarks

We have proposed a new algorithm, EDF-fm, which is based on EDF, for scheduling recurrent soft real-time task systems on multiprocessors, and have derived a tardiness bound that can be guaranteed under it. Our algorithm places no restrictions on the total system utilization, but requires per-task utilizations to be at most one-half of a processor's capacity. This restriction is quite liberal, and hence, our algorithm can be expected to be sufficient for scheduling a large percentage of soft real-time applications. Furthermore, under EDF-fm, only a bounded number of tasks need migrate, and each migrating task will execute on exactly two processors. Thus, task migrations are restricted and the migration overhead of EDF-fm is limited. We have also proposed heuristics for assigning tasks to processors and evaluated them, and proposed the use of the job-slicing technique, when possible, for significantly reducing the actual tardiness observed in practice. Finally, we have presented exponential-time formulas for

[7]Under heavy traffic conditions, the traffic intensity or the average utilization of the processor converges to one.

computing more accurate tardiness bounds, which may be used during offline system design.

We have only taken a first step towards understanding tardiness under EDF-based algorithms on multiprocessors and have not addressed all practical issues concerned. Foremost, the migration overhead of job slicing would translate into inflated execution costs for migrating tasks, and to an eventual loss of schedulable utilization. Hence, an iterative procedure for optimally slicing jobs may be needed. Next, our assumption that arbitrary task assignments are possible may not be true if tasks are not independent. Therefore, given a system specification that includes dependencies among tasks and tardiness that may be tolerated by the different tasks, a framework that determines whether a task assignment that meets the system requirements is feasible, is required. Finally, our algorithm, like every partitioning-based scheme, suffers from the drawback of not being capable of supporting dynamic task systems in which the set of tasks and task parameters can change at runtime. We defer addressing these issues to future work.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, December 1998.

[2] L. Abeni and G. Buttazzo. QoS guarantees using probabilistic deadlines. In *Proceedings of the 11th Euromicro Conference of Real-Time Systems*, pages 242–249, June 1999.

[3] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, July 2005.

[4] B. Andersson, S. Baruah, and J. Jonsson. Static priority scheduling on multiprocessors. In *Proceedings of the 22nd Real-Time Systems Symposium*, pages 193–202, December 2001.

[5] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 33–40, July 2003.

[6] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 123–132, December 1998.

[7] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130, February 2001.

[8] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129, December 2003.

[9] S. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, June 2004.

[10] S. Baruah and J. Carpenter. Multiprocessor fixed-priority scheduling with restricted inter-processor migrations. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 195–202, July 2003.

[11] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[12] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001.

[13] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[14] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice Hall Professional Technical Reference, 1973.

[15] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. In *Proceedings of the 12th International Workshop on Parallel and Distributed Real-Time Systems*, April 2004. 8 pages (on CD-ROM).

[16] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, December 2005.

[17] J. L. Diaz, D. F. Garcia, K. Kim, C.-G. Lee, L. Bello, J. M. Lopez, S. L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 289–300, December 2002.

[18] B. Doytchinov, J. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with Earliest-Deadline-First queue discipline. *Annals of Applied Probability*, 11(2):332–378, 2001.

[19] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[20] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[21] E. D. Jensen, C. D. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[22] A. Khemka and R. K. Shyamasundar. Multiprocessor scheduling of periodic tasks in a hard real-time environment. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 76–81, March 1992.

[23] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 110–117. IEEE, December 1995.

[24] L. Kurk, J. Lehoczky, S. Shreve, and S.-N. Yeung. Earliest-Deadline-First service in heavy-traffic acyclic networks. *Annals of Applied Probability*, 14(3):1306–1352, 2004.

[25] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11st IEEE Real-Time Systems Symposium*, pages 201–209. IEEE, December 1990.

[26] J. P. Lehoczky. Real-time queueing theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 186–195, December 1996.

[27] J. P. Lehoczky, L. Sha, and Y. Ding. Rate-monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[28] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[29] J. W. S. Liu, K.-J. Lin, W.-K.Shih, and A. C. Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991.

[30] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[31] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 25–34, June 2000.

[32] A. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-Time Environments*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1983.

[33] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A.K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2/3):101–155, November/December 2004.

[34] L. Sha and J. Goodenough. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62, 1990.

[35] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, Institut National de Recherche en Informatique et en Automatique, 1996.

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

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

[38] T.-S. Tia, D.-Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the 2nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 164–173, May 1995.

[39] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 311–320, December 2005.

[40] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 239–248. IEEE, December 2000.

Appendix

Lemma 1 *For any two synchronous, periodic tasks T and U that are complementary, a schedule in which every subtask of T is scheduled in the first slot of its window and every subtask of U in its last slot, or vice versa, is feasible on one processor.*

Proof: To show that the lemma holds, it suffices to show the following: for all t, if $r(T_i) = t$ holds for some i, then there does not exist a j such that $d(U_j) = t + 1$. (Here t is a non-negative integer, and i and j are positive integers.) The contrapositive of the above assertion would then imply that for all t, if there exists a j such that $d(U_j) = t + 1$, then there does not exist an i for which $r(T_i) = t$. Hence, a schedule as described in the statement of the lemma would be feasible.

By (3), $r(T_i) = t$ implies that $\left\lfloor \frac{i-1}{wt(T)} \right\rfloor = t$ holds. Therefore,

$$\begin{aligned} & \frac{i-1}{wt(T)} \geq t \\ \Rightarrow\ & i \geq t \cdot wt(T) + 1 \\ \Rightarrow\ & t - i \leq t(1 - wt(T)) - 1 \\ \Rightarrow\ & \frac{t-i+1}{1-wt(T)} \leq t. \end{aligned} \tag{26}$$

Furthermore, by $\left\lfloor \frac{i-1}{wt(T)} \right\rfloor = t$, we also have the following.

$$
\begin{aligned}
& \frac{i-1}{wt(T)} \;<\; t+1 \\
\Rightarrow\;\; & i-1 \;<\; wt(T) + t \cdot wt(T) \\
\Rightarrow\;\; & t-i+1+wt(T) \;>\; t(1-wt(T)) \\
\Rightarrow\;\; & \frac{t-i+1+wt(T)}{1-wt(T)} \;>\; t \\
\Rightarrow\;\; & \frac{t-i+1+wt(T)}{1-wt(T)} + \frac{1-wt(T)}{1-wt(T)} \;>\; t + \frac{1-wt(T)}{1-wt(T)} \\
\Rightarrow\;\; & \frac{t-i+2}{1-wt(T)} \;>\; t+1 \qquad (27)
\end{aligned}
$$

By (26) and (27), it follows that there does not exist an integer j for which $\left\lceil \frac{j}{1-wt(T)} \right\rceil = t+1$ holds. By Definition 1, $wt(U) = 1 - wt(T)$. Therefore, by (4), it follows that there does not exist a subtask U_j such that $d(U_j) = t+1$ holds. ■