# Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors

by
Anand Srinivasan

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2003

Approved by:

James H. Anderson, Advisor

Sanjoy K. Baruah, Reader

Kevin Jeffay, Reader

Giorgio Buttazzo, Reader

Prasun Dewan, Reader

Ketan Mayer-Patel, Reader

## ABSTRACT

**Anand Srinivasan**

**Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors**

**(Under the direction of James H. Anderson)**

Proportionate fair (Pfair) scheduling is the only known way to optimally schedule periodic real-time task systems on multiprocessors in an on-line manner. Under Pfair scheduling, the execution of each task is broken into a sequence of quantum-length *subtasks* that must execute within intervals of approximately-equal lengths. This scheduling policy results in allocations that mimic those of an ideal "fluid" scheduler, and in periodic task systems, ensures that all deadlines are met.

Though Pfair scheduling algorithms hold much promise, prior to our work, research on this topic was limited in that only static systems consisting of synchronous periodic tasks were considered. My dissertation thesis is that *the Pfair scheduling framework for the on-line scheduling of real-time tasks on multiprocessors can be made more flexible by allowing the underlying task model to be more general than the periodic model and by allowing dynamic task behaviors. Further, this flexibility can be efficiently achieved.*

Towards the goal of improving the efficiency of Pfair scheduling algorithms, we develop the $PD^2$ Pfair algorithm, which is the most efficient optimal Pfair scheduling algorithm devised to date. Through a series of counterexamples, we show that it is unlikely that a more efficient optimal Pfair algorithm exists. We also introduce the concept of ERfair scheduling, which is a work-conserving extension of Pfair scheduling. In addition, we study the non-optimal earliest-pseudo-deadline-first (EPDF) Pfair algorithm, which is more efficient than $PD^2$, and present several scenarios under which it is preferable to $PD^2$.

We address the flexibility issue by developing the intra-sporadic (IS) task model and by considering the scheduling of dynamic task systems. The well-known sporadic model generalizes the periodic model by allowing jobs to be released late. The IS model generalizes this notion further by allowing late as well as early subtask releases. Such a generalization is useful for modeling applications in which the instantaneous rate of releases differs greatly from the average rate of releases (*e.g.*, an application that

receives packets over a network). We prove that PD$^2$ is optimal for scheduling static IS task systems on multiprocessors. In dynamic task systems, tasks are allowed to join and leave, *i.e.*, the set of tasks is allowed to change. This flexibility also allows us to model systems in which the weights of tasks may change. We present sufficient conditions under which joins and leaves can occur under PD$^2$ without causing missed deadlines. Further, we show that these conditions are tight.

Finally, we also provide schemes for multiplexing the scheduling of aperiodic tasks and real-time IS tasks. These approaches aim at improving the response times of aperiodic tasks while ensuring that the real-time IS tasks meet their deadlines. We also provide bounds on aperiodic response times under these schemes; these bounds can be used to obtain admission-control tests for aperiodic tasks with deadlines.

# ACKNOWLEDGMENTS

I want to thank my advisor, Jim Anderson, for his support, enthusiasm, and patience. I have learnt an enormous amount from working with him, and have thoroughly enjoyed doing so. I am also grateful to him for arranging financial support for me throughout my stay at UNC.

Many thanks to Sanjoy Baruah for giving me the opportunity to collaborate with him, and for many lively discussions from which my work has benefited tremendously. Thanks, too, to the rest of my committee: Kevin Jeffay, Prasun Dewan, Giorgio Buttazzo, and Ketan Mayer-Patel. I greatly appreciate their willingness to bend their schedules to accommodate the myriad meetings and exams, and also for being supportive and encouraging of my work. I am also grateful to them for enduring much of this pain by means of phone and video conference.

In addition, I would like to thank all my colleagues over the past few years who have helped me become a better researcher. The list would definitely include the following people: Aaron Block, John Carpenter, Uma Devi, Shelby Funk, Phil Holman, Mark Moir, Srikanth Ramamurthy, Jasleen Sahni, and Montek Singh. I am also grateful to the staff of the computer science department at UNC for helping me in innumerable ways and for being a pleasure to work with. Life would be much more difficult (and no fun) without the support and companionship of many good friends. Thanks to all my fellow graduate students, especially, Vibhor, Anu, Ajith, Bala, Gopi, and Aditi for making my stay at UNC a very memorable one.

I thank God for providing me the love and support of a wonderful family. I am grateful to them for encouraging my education from a young age, and allowing my inquisitive nature to blossom. I am also very fortunate to have wonderful in-laws who encouraged me at every step of my Ph.D. Finally, I am forever indebted to my wife Chandna: without her love and constant encouragement, none of this would have been possible. She has been incredibly supportive of me through the ups and downs over the past few years. I cannot thank her enough for being such a wonderful wife and friend.

# Contents

# List of Figures

# Chapter 1

# Introduction

A *real-time system* has two notions of correctness: *logical* and *temporal*. In particular, in addition to producing correct outputs (logical correctness), such a system needs to ensure that these outputs are produced at the correct time (temporal correctness). As an example, consider a computer-controlled robot that is designed to pick up objects from a moving conveyor belt. If the robot moves too early, then the next object may not have arrived and may get blocked by the robot; on the other hand, if it moves late, then it may completely miss the object. The speed of the conveyor belt imposes timing constraints on the operations performed by the robot, and hence it is a real-time system.

Selecting appropriate methods for *scheduling* activities is one of the important considerations in the design of a real-time system; such methods are essential to ensure that all activities are able to meet their timing constraints. The timing constraint of an operation is usually specified using a *deadline*, which corresponds to the time by which that operation must complete. Real-time systems can be broadly classified as *hard* or *soft* depending on the criticality of deadlines. In a hard real-time system, all deadlines must be met; equivalently, a deadline miss results in an incorrect system. For example, the robot described earlier would be a hard real-time system if not picking up an object leads to an incorrect operation such as a chemical spill or a complete halting of the assembly line. Other examples of hard real-time systems include fly-by-wire controllers for airplanes, monitoring systems for nuclear reactors, and automotive braking systems. On the other hand, in a soft real-time system, timing constraints are less stringent; occasional deadline misses do not affect the correctness of the system. Multimedia and gaming applications are very common examples of soft real-time systems. The robot example above may be categorized as a soft real-time system if a timing error only leads to a slight drop in productivity. However, note that it might still be beneficial to view the robot as a hard real-time system because timing requirements are easier to specify in such a system. In particular, if the robot were to be implemented as a soft real-time system, then in addition to timing constraints, we would need to specify the level of productivity loss that is acceptable, and guarantee that this specification is met.

There are several emerging real-time applications that are very complex and have high computational requirements. Examples of such systems include automatic tracking systems and telepresence systems. These applications have timing constraints that are used to ensure high system fidelity and responsiveness, and may also be crucial for correctness in certain applications such as telesurgery. Also, their processing requirements may easily exceed the capacity of a single processor, and a multiprocessor may be necessary to achieve an effective implementation. In addition, multiprocessors are more cost-effective than a single processor of the same capacity because the cost (monetary) of a $k$-processor system is significantly less than that of a processor that is $k$ times as fast (if a processor of that speed is indeed available) [WH95].

The above observations clearly underscore the growing importance of multiprocessors in real-time systems. In this dissertation, we focus on several fundamental issues pertaining to the scheduling of multiprocessor real-time systems. Before discussing the contributions of this dissertation in more detail, we briefly describe some basic concepts pertaining to real-time systems.

## 1.1 Background on Real-time Systems

A real-time system is typically composed of several (sequential) processes with timing constraints. We refer to these processes as *tasks*. In most real-time systems, tasks are *recurrent*, *i.e.*, each task is invoked repeatedly. The *periodic* task model of Liu and Layland [LL73] provides the simplest notion of a recurrent task. Each periodic task $T$ is characterized by a *phase* $T.\phi$, a *period* $T.p$, a *relative deadline* $T.d$, and an *execution requirement* $T.e$ ($\leq T.d$). Such a task is invoked every $T.p$ time units, with its first invocation occurring at time $T.\phi$. We refer to each invocation of a task as a *job*, and the corresponding time of invocation as the job's *release time*. Each job of $T$ requires at most $T.e$ units of processor time to complete, and it must complete within $T.d$ time units after its release. Thus, the relative deadline parameter is used to specify the timing constraints of the jobs of a periodic task. Unless stated otherwise, we assume in this dissertation that $T.d = T.p$, *i.e.*, the relative deadline of a periodic task equals

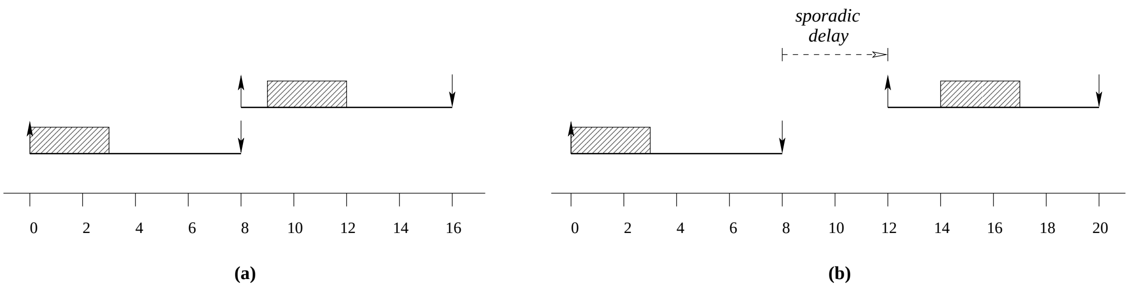



Figure 1.1: Each job is shown on a separate line. The up arrows corresponds to job releases and down arrows correspond to job deadlines. Inset **(a)** shows a periodic task $T$ with $T.\phi = 0$, $T.e = 3$, $T.p = 8$, $T.d = 8$. Inset **(b)** shows a sporadic task with the same parameters; the second job is released four time units late.

its period. In other words, each job must complete before the release of the next job of the same task.

As an example of a periodic task, consider the application controlling the robot in the example described earlier. Since the robot periodically picks an object from the conveyor belt, the controller application consists of several periodic tasks. Most of these tasks will be in one of two categories: tasks that process information received from the robot such as its position, velocity, orientation, *etc.*, and tasks that generate signals sent to the robot to change its velocity or direction, if necessary. Note that the second type of task may depend on the information processed by the first type; such a dependency can be enforced by making the phase of the dependent task equal to the period of the task on which it depends.

While the periodic task model can model time-driven processes effectively, it may not be sufficient to model processes that are driven by external interrupts occurring at *approximately* periodic intervals. In the *sporadic* task model, the notion of a periodic task is generalized by allowing the time interval between consecutive invocations of a task to be more than the task's period. In other words, $T.p$ refers to the *minimum separation* between consecutive job releases of task $T$. Figure 1.1 illustrates the difference between a periodic and a sporadic task.

A *periodic task system* is comprised of a collection of periodic tasks. (A *sporadic task system* is defined similarly.) A periodic task system in which all tasks have a phase of zero is called a *synchronous* periodic task system. The *weight* or *utilization*[1] of a task $T$, denoted $wt(T)$, is the ratio of its execution requirement to its period, *i.e.*,

[1]We use the terms "weight" and "utilization" interchangeably in this dissertation.

$wt(T) = \frac{T.e}{T.p}$. Because $0 < T.e \leq T.p$, we have $0 < wt(T) \leq 1$. The weight of a task determines the fraction (*i.e.*, *share*) of a single processor that it requires. For example, the task shown in Figure 1.1 has a weight of 3/8, *i.e.*, it requires a processor share of 37.5% to meet its timing constraints. The *weight* (or *utilization*) *of a task system* is the sum of the weights of all tasks in the system.

In this dissertation, we assume that all tasks are *preemptive*, *i.e.*, a task can be interrupted during its execution and resumed later from the same point. Unless stated otherwise, we assume that the overhead of a preemption is zero. We further assume that all tasks are *independent*, *i.e.*, the execution of a task is not affected by the execution of other tasks. In particular, we assume that tasks do not share any resources other than the processor, and that they do not self-suspend during execution.

**Feasibility and optimality.** A task system $\tau$ is said to be *feasible* if there exists a schedule for $\tau$ in which each job released by a task in $\tau$ meets its deadline. A *feasibility test* for a class of task systems is specified by giving a condition that is necessary and sufficient to ensure that any task system in that class is feasible.

The algorithm that is used to schedule tasks (*i.e.*, allocate processor time to tasks) is referred to as a *scheduling algorithm*. A task system $\tau$ is said to be *schedulable* by algorithm $A$ if $A$ can guarantee the deadlines of all jobs of every task in $\tau$. A condition under which all task systems within a class of task systems are schedulable by $A$ is referred to as a *schedulability test* for $A$ for that class of task systems. A scheduling algorithm is defined as *optimal* for a class of task systems if its schedulability condition is identical to the feasibility condition for that class.

**On-line versus off-line scheduling.** In *off-line* scheduling, the entire schedule for a task system (up to a certain time such as the the least common multiple (LCM) of all task periods) is pre-computed before the system actually runs; the actual run-time scheduling is done using a table based on this pre-computed schedule. On the other hand, an *on-line* scheduler selects a job for scheduling without any knowledge of future job releases. (Note that an on-line scheduling algorithm can also be used to produce an off-line schedule.) Clearly, off-line scheduling is more efficient at run-time; however, this efficiency comes at the cost of flexibility. In order to produce an off-line schedule, it is necessary to know the exact release times for all jobs in the system. However, such knowledge may not be available in many systems, in particular, those consisting of sporadic tasks, or periodic tasks with unknown phases. Even if such knowledge is available, then it may be impractical to store the entire pre-computed schedule (*e.g.*, if the LCM of the task periods is very large). On the other hand, on-line schedulers need to be very efficient, and hence, may need to make sub-optimal scheduling decisions, resulting in schedulability loss. In this dissertation, we focus only on on-line scheduling algorithms.

**Static versus dynamic priorities.** Most scheduling algorithms are priority-based: they assign priorities to the tasks or jobs in the system and these priorities are used to select a job for execution whenever scheduling decisions are made. A priority-based scheduling algorithm can determine task or job priorities in different ways.

A scheduling algorithm is called a *static-priority* algorithm if there is a unique priority associated with each task, and all jobs generated by a task have the priority associated with that task. Thus, if task $T$ has higher priority than task $U$, then whenever both have active jobs, $T$'s job has higher priority than $U$'s job. An example of a scheduling algorithm that uses static priorities is the rate-monotonic (RM) algorithm [LL73]. The RM algorithm assigns higher priority to tasks with shorter periods. Liu and Layland [LL73] proved that RM is optimal among all static-priority algorithms for scheduling periodic tasks on uniprocessors.

*Dynamic-priority* algorithms allow more flexibility in priority assignments; a task's priority may vary across jobs or even within a job. An example of a scheduling algorithm that uses dynamic priorities is the earliest-deadline-first (EDF) algorithm [LL73]. EDF assigns higher priority to jobs with earlier deadlines, and has been shown to be optimal for scheduling periodic and sporadic tasks on uniprocessors [LL73, Mok83]. The least-laxity-first (LLF) algorithm [Mok83] is also an example of a dynamic-priority algorithm that is optimal on uniprocessors. The *laxity* of a job at time $t$ is $d - t - e$, where $d$ is the job's deadline and $e$ is the job's remaining execution time at time $t$. As its name suggests, under LLF, jobs with lower laxity are assigned higher priority.

Figure 1.2 illustrates the behavior of the EDF and RM algorithms on a uniprocessor for two tasks $T$ and $U$ with the following parameters: $T.\phi = U.\phi = 0$, $T.e = 3$, $T.p = 5$, $U.e = 1$, and $U.p = 3$. Inset (a) shows the schedule produced by the RM algorithm. RM assigns higher priority to task $U$ because it has a smaller period. At time 1, $U$'s job completes and $T$'s job is scheduled. $U$'s second job, which is released at time 3, preempts $T$'s job. At time 4, $U$'s job completes execution and $T$'s job resumes execution. $T$'s job finally completes at time 5. Since $U$ has no unfinished jobs at time 5, when $T$ releases a new job at time 5, RM schedules $T$ again. By reasoning in a similar

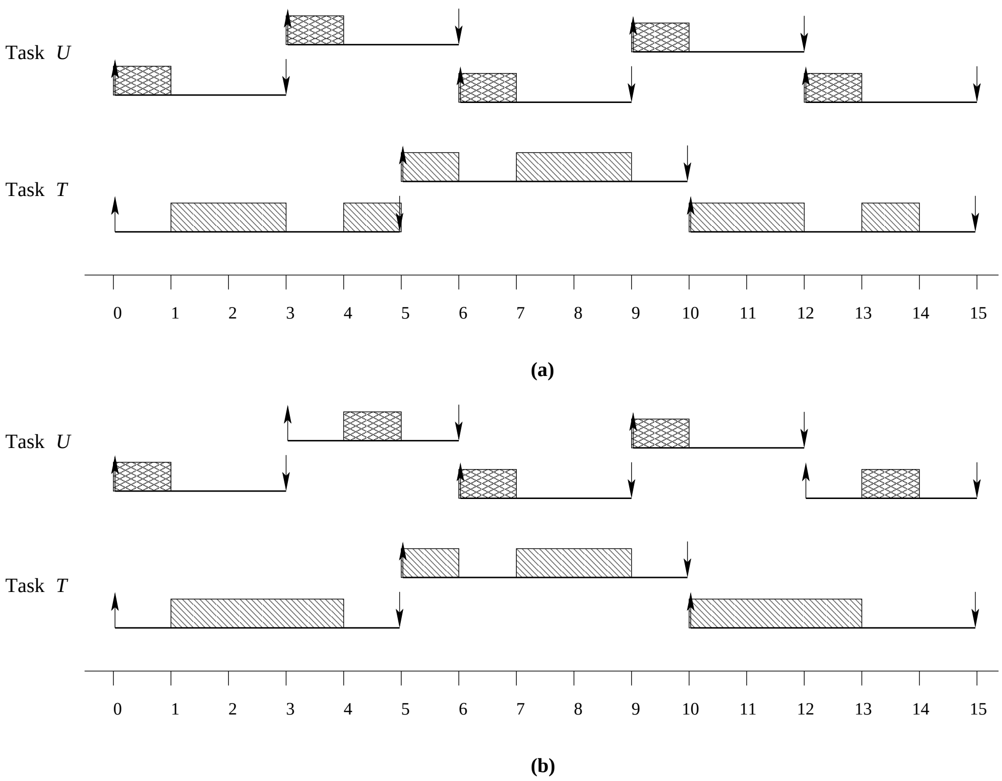



Figure 1.2: The notation used in this figure is the same as in Figure 1.1. Insets **(a)** and **(b)**, respectively, show the RM and EDF schedules for the following periodic task system consisting of two tasks $T$ and $U$: $T.\phi = U.\phi = 0$, $T.e = 3$, $T.p = 5$, $U.e = 1$, and $U.p = 3$. This pattern of job releases repeats after time 15, and hence, the schedule also repeats.

manner, we can show that the rest of the schedule is as illustrated in Figure 1.2(a).

Inset (b) shows the schedule produced by the EDF algorithm. At time 0, $U$'s job has a deadline at time 3, while $T$'s job has a deadline at time 5. Therefore, EDF schedules $U$'s job at time 0. At time 1, $U$'s job finishes execution and $T$'s job is scheduled by EDF. At time 3, $U$ releases a new job that has a deadline at time 6, which is greater than the deadline of $T$'s job. Therefore, EDF continues executing $T$ until time 4, at which time $U$ is scheduled. By reasoning in a similar manner, we can show that the rest of the schedule is as illustrated in Figure 1.2(b).

The set of dynamic-priority algorithms is a super-set of the set of static-priority algorithms because every static-priority scheduling algorithm is also a dynamic-priority algorithm. Static-priority algorithms are more efficient as they incur lower overhead while making scheduling decisions; however, there exist task systems that can be cor-

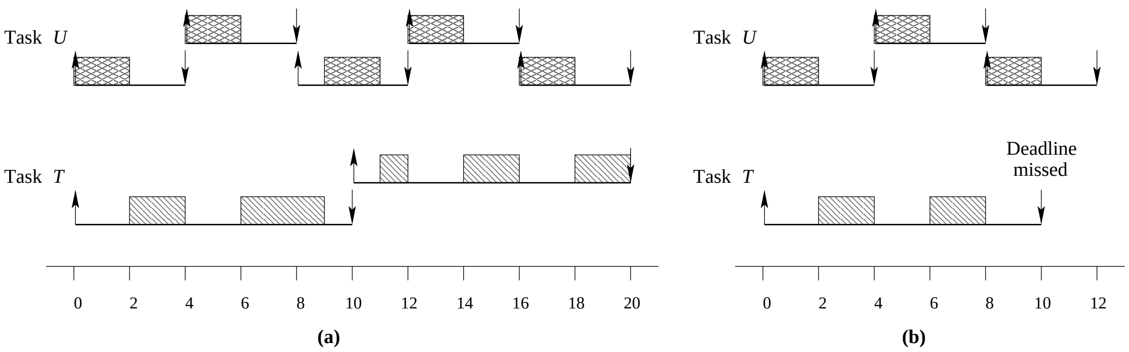



Figure 1.3: Insets **(a)** and **(b)**, respectively, show the EDF and RM schedules for the following periodic task system consisting of two tasks $T$ and $U$: $T.\phi = U.\phi = 0$, $T.e = 5$, $T.p = 10$, $U.e = 2$, and $U.p = 4$. The first job of task $T$ misses its deadline at time 10.

rectly scheduled if and only if dynamic priorities are allowed. For example, consider the following periodic task system with two tasks $T$ and $U$: $T.e = 5, T.p = 10$ and $U.e = 2, U.p = 4$. Figure 1.3(a) shows the schedule under EDF, and Figure 1.3(b) shows the schedule under RM — a deadline is missed at time 10 under RM. Since RM is an optimal static-priority algorithm on uniprocessors, no other static-priority algorithm can correctly schedule this task system. (In particular, if $T$ is assigned higher priority, then $U$'s first job will miss its deadline at time 4.)

## 1.2 Real-time Scheduling on Multiprocessors

As mentioned earlier, our focus in this dissertation is the scheduling of multiprocessor real-time systems. We now give a brief introduction to prior research in this area.

In this dissertation, we assume that all tasks are sequential and can execute on only one processor at a time. However, tasks are allowed to be preempted, and resumed at a later time, maybe on a different processor. In other words, a task is also allowed to *migrate* to a different processor. In general, there are two approaches for scheduling on multiprocessors depending on whether task migration is allowed: *partitioning* and *global scheduling*.

Under partitioning, the workload is partitioned among the available processors and tasks are then scheduled on a per-processor basis. Thus, a task is assigned to a particular processor and always executed on that processor. On the other hand, under global scheduling, all eligible tasks are placed in a single ready queue, and the sched-

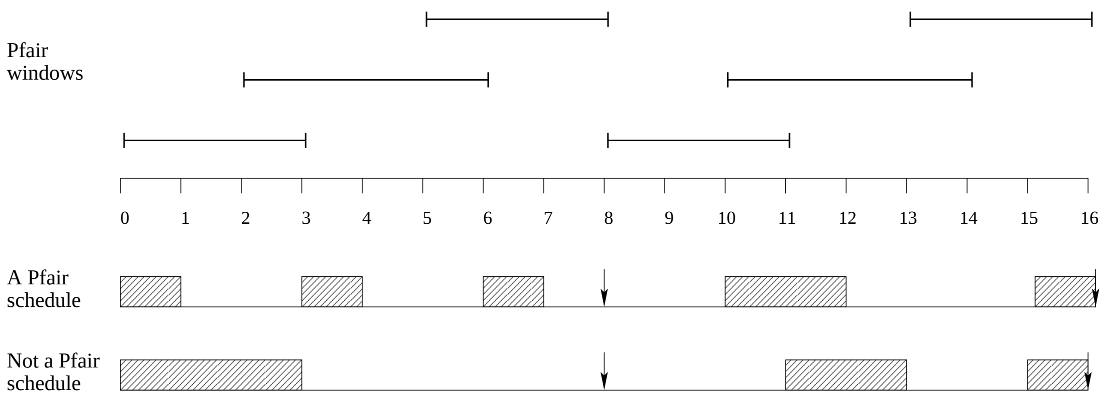



Figure 1.4: The Pfair windows of the first two jobs (or six subtasks) of a task $T$ such $T.e = 3$ and $T.p = 8$. During each job of $T$, each of the three subtasks must be executed during its window in order to satisfy the Pfairness rate constraint.

uler selects the highest-priority tasks from this queue for execution. Thus, under global scheduling, a task is not fixed to a processor and is allowed to migrate. Until recently, no optimal polynomial-time global algorithm for scheduling real-time recurrent tasks was known; it was also not known whether such an algorithm even exists. The seminal work of Baruah, Cohen, Plaxton, and Varvel [BCPV96] on *proportionate fair* (*Pfair*) scheduling proved that periodic task systems could be optimally scheduled on-line in polynomial time using global scheduling algorithms based on Pfairness.

Pfair scheduling differs from more conventional real-time scheduling approaches in that tasks are explicitly required to execute at steady rates. In most real-time scheduling disciplines, the notion of a rate is implicit. For example, in an EDF or RM schedule, a task $T$ executes at a rate defined by its required utilization ($T.e/T.p$) over large intervals. However, its execution rate over short intervals, *e.g.*, individual periods, may vary significantly. Hence, the notion of a rate under the periodic task model is a bit inexact.

Under Pfair scheduling, each task is executed at an approximately-uniform rate by breaking it into a series of quantum-length *subtasks*. Each period of a task is then subdivided into a sequence of (potentially overlapping) subintervals of approximately equal lengths, called *windows*. To satisfy the Pfairness rate constraint, each subtask must execute within its associated window. (Refer to Figure 1.4.) Pfair scheduling algorithms typically assign priorities to subtasks and try to ensure that each subtask is scheduled in its window; this, in turn, is sufficient to guarantee that each job meets its deadline.

By breaking tasks into uniform-sized subtasks, Pfair scheduling circumvents many of the bin-packing-like problems that lie at the heart of intractability results pertaining to multiprocessor scheduling. Indeed, Pfair scheduling is presently the only known approach for optimally scheduling periodic tasks on multiprocessors. Baruah *et al.* presented two optimal Pfair algorithms, PF [BCPV96] and PD [BGP95]; PD is the more efficient of the two. (Baruah [Bar95] also presented a non-optimal static-priority Pfair algorithm called the weight-monotonic (WM) algorithm.)

In spite of the optimality of Pfair scheduling algorithms, partitioning is currently the favored approach. There are three main reasons for this. First, though partitioning approaches are not theoretically optimal, they tend to perform well in practice. Second, Pfair scheduling algorithms can result in frequent preemptions and migrations resulting in excessive overhead. Third, until our work, all work on Pfair scheduling was limited to periodic tasks, and it was not known whether more efficient Pfair schedulers could be obtained.

However, some of these reasons for always favoring partitioning are no longer valid. In particular, many emerging real-time applications are highly *dynamic*: the set of tasks in the system may change or their timing requirements may change. Partitioning is not suitable for such systems because any change in the set of tasks may lead to a re-partitioning of the entire system causing unacceptable overhead. Further, recent architectural advances have led to the development of multiprocessors with low migration overheads (*e.g.*, single chip multiprocessors [FGP$^+$00]).

Finally, as we demonstrate in this dissertation, Pfair scheduling algorithms can be made more efficient and flexible. Experimental results also indicate that the performance of these algorithms is comparable to partitioning approaches, even for static task systems.

## 1.3 Contributions

The main thesis supported by this dissertation is the following.

> *The Pfair scheduling framework for the on-line scheduling of real-time tasks on multiprocessors can be made more flexible by allowing the underlying task model to be more general than the periodic model and by allowing the task system to be dynamic. Further, this flexibility can be efficiently achieved.*

In the following subsections, we describe the contributions of this dissertation in more detail. In addition to efficiency (which is clearly important), flexibility is essential in several of the emerging real-time applications as described earlier. In this dissertation, we address the flexibility issue by developing the intra-sporadic (IS) task model (briefly described in Section 1.3.3) and by considering the scheduling of dynamic task systems (Section 1.3.4). The IS model is a generalization of the sporadic task model. We also provide schemes to improve response times of non-recurrent real-time tasks while ensuring deadlines of the real-time tasks (Section 1.3.6). Towards the goal of improving the efficiency of Pfair scheduling algorithms, we develop the $PD^2$ Pfair algorithm (Section 1.3.1), which is the most efficient optimal Pfair scheduling algorithm devised to date, and the concept of ERfair scheduling (Section 1.3.2). We also study the simpler (non-optimal) earliest-pseudo-deadline-first (EPDF) Pfair algorithm, and present several scenarios under which it is preferable to other Pfair algorithms (Section 1.3.5).

### 1.3.1 The $PD^2$ Pfair Algorithm

The $PD^2$ algorithm was obtained by simplifying the priority definition in the PD algorithm of Baruah *et al.* [BGP95]. We prove that $PD^2$ is optimal for scheduling periodic tasks on multiprocessors; it is currently the most efficient among all known optimal Pfair scheduling algorithms. We also show through several counterexamples that the priority definition in $PD^2$ cannot be simplified further without sacrificing optimality. These counterexamples illustrate the minimality of the $PD^2$ priority definition.

As mentioned earlier, Pfair scheduling can lead to frequent preemptions and migrations. In order to study the effect of preemption and migration overheads on the performance of $PD^2$, we conducted experiments comparing $PD^2$ to a partitioning approach. Our results demonstrate that $PD^2$ performs competitively because the schedulability loss due to these overheads is offset by the fact that $PD^2$ provides much better analytical bounds on schedulability than partitioning.

### 1.3.2 Early-release Fair Scheduling

One undesirable characteristic of Pfair scheduling is that jobs can be ineligible according to the Pfairness rate constraint, despite being ready. Consequently, processors may be idle while ready unscheduled jobs exist. In other words, Pfair scheduling algorithms are not "work-conserving." (An algorithm is *work-conserving* if it never leaves a processor idle while an active job exists.) To address this problem, we propose a work-conserving variant of Pfair scheduling called *early-release fair* (*ERfair*) scheduling. The ERfair version of the $PD^2$ algorithm incurs lower run-time overhead than the Pfair version. Further, as we show, job response times are likely to be much better under ERfair scheduling.

### 1.3.3 The Intra-sporadic Task Model

Though the periodic and sporadic task models are the most widely studied, they are not applicable in all situations. Consider a multimedia application receiving packets over a network. Since packet arrivals may be late or bursty, these arrivals may not have a periodic or sporadic pattern. To enable applications to handle such scenarios more seamlessly, we present a new task model called the intra-sporadic (IS) model, which treats burstiness and late arrivals as first-class concepts. We also analytically prove that $PD^2$ is optimal for scheduling IS task systems. To the best of our knowledge, this is the *first* work on optimal algorithms for scheduling such tasks on multiprocessors. Further, since the IS model generalizes the notion of a sporadic model, it follows that $PD^2$ is optimal for scheduling sporadic tasks on multiprocessors as well.

### 1.3.4 Scheduling of Dynamic Task Systems

With the proliferation of multimedia, gaming, and virtual-reality applications, dynamic real-time task systems are becoming increasingly important. In such systems, tasks may be initiated at arbitrary times and may execute for a finite duration. To support such tasks, mechanisms are needed for handling task "joins" and "leaves" without adversely affecting the timeliness of other tasks. Dynamic task systems have been well-studied in work on uniprocessors [BGP+97, SRLR89, SAWJ+96, TBW92]. In particular, necessary and sufficient conditions for task joins and leaves have been presented for fair-scheduled uniprocessor systems [BGP+97, SAWJ+96]. We extend this work to multiprocessor Pfair-scheduled systems. We show that the multiprocessor variants of the uniprocessor conditions are insufficient, in general, when any priority-based Pfair scheduling algorithm is used. We also derive sufficient join/leave conditions for $PD^2$-scheduled systems, and demonstrate their tightness through counterexamples.

### 1.3.5 Scheduling of Soft Real-time Multiprocessor Systems

Since occasional deadline misses are allowed in soft real-time systems, PD$^2$ can be simplified without a large loss in system performance. We consider the scheduling of real-time multiprocessor systems using the EPDF algorithm, which uses a more efficient (but non-optimal) method to determine subtask priorities. In particular, we derive a condition for scheduling using EPDF that ensures that deadlines are missed by at most one quantum. This condition is very liberal and should often hold in practice. We also generalize this by deriving conditions under which EPDF guarantees a given bound on *tardiness*, *i.e.*, the amount by which a deadline is missed. Additionally, we present conditions under which EPDF can guarantee all deadlines, thus allowing us to identify those *hard* real-time systems in which EPDF can be successfully used.

### 1.3.6 Scheduling of Aperiodic Tasks

Interrupt handling routines that are invoked infrequently may be modeled as *aperiodic* tasks. Unlike periodic and sporadic tasks, such tasks release only a single job, and may or may not have deadlines. We present several approaches for scheduling aperiodic tasks along with hard real-time IS tasks. In these approaches, the spare processor capacity (*i.e.*, the difference between number of processors and the total weight of the real-time IS tasks) is distributed among several aperiodic servers in a greedy manner. Each server is scheduled as an IS task and it, in turn, schedules the aperiodic tasks that are assigned to it. We derive bounds on response times of the aperiodic tasks; these bounds can also be used to design admission control tests for aperiodic tasks with deadlines.

## 1.4 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we describe prior work on scheduling in multiprocessor real-time systems, and in particular, prior work on Pfair scheduling. In Chapter 3, we describe the PD$^2$ algorithm, the concept of ERfair scheduling, and also prove the optimality of PD$^2$ for both Pfair- and ERfair-scheduled systems. In Chapter 4, we describe the IS task model, and present sufficient join/leave conditions for dynamic IS task systems scheduled using PD$^2$. Chapter 5 presents results concerning the EPDF algorithm, and Chapter 6 describes the aperiodic server approaches discussed above. Finally, in Chapter 7, we summarize our contributions and discuss directions for future research.

# Chapter 2

# Background and Related Work

In this chapter, we survey prior research on the scheduling of real-time tasks on multiprocessors. We begin by stating an impossibility result pertaining to the on-line scheduling of hard real-time aperiodic tasks on multiprocessors.

**Theorem 2.1 (Dertouzos and Mok [DM89]).** *No scheduling algorithm is optimal for scheduling hard real-time aperiodic tasks on two or more processors if all release times, deadlines, and execution requirements are not known a priori.*

This is in sharp contrast to on-line scheduling on a uniprocessor, for which the earliest-deadline-first (EDF) algorithm has been shown to be optimal [Der74, LL73]. Though Theorem 2.1 does not apply to the scheduling of recurrent tasks, it does indicate the difficulty of obtaining optimal on-line scheduling algorithms for multiprocessors.

As mentioned in Section 1.2, scheduling approaches on multiprocessors fall into one of two categories: *partitioning* and *global scheduling*. We first briefly describe the basic concepts behind each approach and then describe in more detail relevant results about scheduling algorithms and their schedulability conditions.

**Partitioning.** In partitioning schemes, each processor schedules tasks independently from a local ready queue. When a new task arrives, it is assigned to one of these ready queues and executes only on the associated processor. The main advantage of partitioning approaches is that they reduce a multiprocessor scheduling problem to a set of uniprocessor ones. Unfortunately, partitioning has two negative consequences. First, finding an optimal assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense [GJ79]. Thus, tasks are usually partitioned using non-optimal heuristics. Second, task systems exist that are schedulable (*i.e.*, their

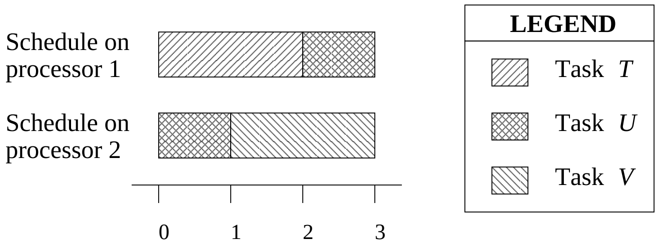



Figure 2.1: A schedule that allows migration for a two-processor system with three tasks each of weight 2/3. The schedule repeats after time 3. The schedule on each processor is shown on separate lines. Note that the job of task $U$ must be allowed to migrate as shown to meet its deadline at time 3.

deadlines can be guaranteed) if and only if tasks are *not* partitioned. For instance, consider a system of three tasks each with an execution requirement of 2 and a period of 3 to be scheduled on two processors. There is no way to partition these three tasks into two sets such that the total utilization of each set is at most one. On the other hand, as Figure 2.1 illustrates, these tasks can be scheduled on two processors if tasks are allowed to migrate.

**Global scheduling.** In contrast to partitioning, under global scheduling, all ready tasks are stored in a single priority-ordered queue. Since a single system-wide priority space is assumed, the highest-priority task is selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled. Whereas partitioning avoids migration, global scheduling may result in frequent migrations due to the use of a shared queue. Dhall and Liu [DL78] showed that global scheduling with optimal *uniprocessor* scheduling algorithms, such as EDF and RM, may result in arbitrarily low processor utilization.

To see why, consider the following synchronous periodic task system to be scheduled on $M$ processors: $M$ tasks with period $p$ and execution requirement 2, and one task with period $p+1$ and execution requirement $p$. At time 0, both EDF and RM favor the $M$ tasks with period $p$. The task with period $p+1$ does not get scheduled until time 2, by when its deadline cannot be guaranteed. (Figure 2.2 illustrates this for $M = 2$ and $p = 5$.) Note that the total utilization of this task system is $\frac{2M}{p} + \frac{p}{p+1}$; as $p$ tends to $\infty$, this value tends to 1 from above. Thus, we have the following result.

**Theorem 2.2 (Dhall and Liu [DL78]).** *EDF and RM cannot correctly schedule all task systems $\tau$ with $\sum_{T \in \tau} wt(T) \leq B$ on $M$ processors, for any $B > 1$.*

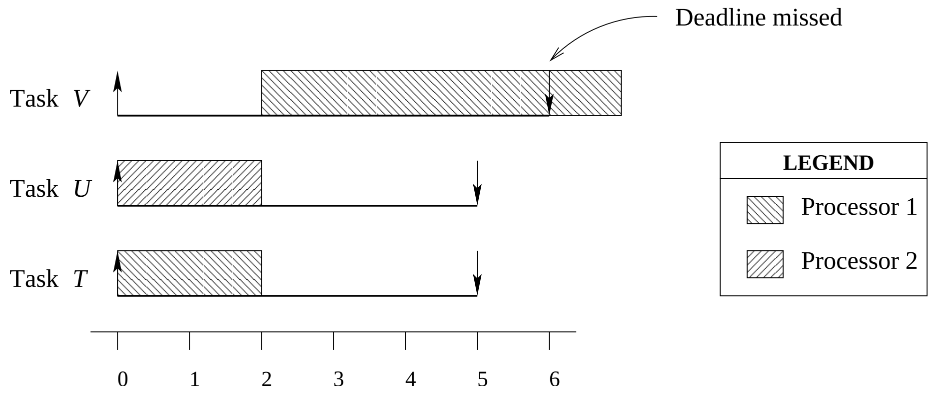



Figure 2.2: Tasks $T$, $U$, and $V$ have the following parameters: $T.e = U.e = 2$, $T.p = U.p = 5$, $V.e = 5$, and $V.p = 6$. Jobs of each task are shown on a separate line. Under both EDF and RM, jobs of tasks $T$ and $U$ are scheduled at time 0. This causes $V$'s job to miss its deadline.

Despite this negative result, recent research on Pfair scheduling algorithms has shown considerable promise for the development of efficient and flexible global scheduling algorithms.

In Section 2.1, we describe research on partitioning in more detail; later, in Section 2.2, we introduce some of the basic concepts of Pfair scheduling.

## 2.1 Partitioning

Under partitioning, each task is assigned to a processor, on which it exclusively executes. The primary advantage of partitioning is that, once tasks are assigned to processors, each processor can be scheduled independently using uniprocessor scheduling algorithms. Before considering partitioning approaches in detail, we give a brief overview of relevant research on uniprocessors.

### 2.1.1 Schedulability Results for Uniprocessor Systems

We first present known schedulability tests for RM and then present similar results for EDF.

**Schedulability conditions for RM.** Liu and Layland [LL73] proved that RM is optimal among all static-priority algorithms for scheduling synchronous periodic task systems on uniprocessors. In other words, if there exists some static-priority scheduling algorithm that can correctly schedule a given task system, then RM can also correctly schedule it. They also showed that RM can schedule any task system $\tau$ that satisfies the following condition.

$$\sum_{T \in \tau} wt(T) \leq N(2^{\frac{1}{N}} - 1) \tag{2.1}$$

The right-hand-side of (2.1) converges to 0.69 from above as $N$ tends to $\infty$. Thus, a periodic task system is schedulable by RM if its total weight (*i.e.*, utilization) is at most 0.69. However, this test is only sufficient and not necessary, *i.e.*, there exist task systems with larger utilizations that can be correctly scheduled by RM. Lehoczky, Sha, and Ding [LSD89] presented the following necessary and sufficient schedulability test for RM. Let $H(T)$ denote the set of tasks that have periods at least $T.p$, *i.e.*, $H(T)$ consists of the tasks that may be assigned higher priority than $T$. Then, RM can correctly schedule a synchronous periodic task system $\tau$ if and only if the following condition holds.

$$(\forall T \in \tau,\ \exists t \in \{0, \ldots, T.p - 1\} : T.e + \sum_{U \in H(T)} \left\lceil \frac{t}{U.p} \right\rceil \cdot U.e \leq t) \tag{2.2}$$

Note that though this test improves upon (2.1), its time complexity is higher. In particular, the condition in (2.1) can be verified in $O(N)$ (*i.e.*, polynomial) time, whereas the condition in (2.2) takes $O(N \times max_{T \in \tau} T.p)$ time (which is pseudo-polynomial in the size of the bit-representation of the input).

**Schedulability conditions for EDF.** Liu and Layland [LL73] proved that the following is a feasibility condition for periodic task systems on uniprocessors.

$$\sum_{T \in \tau} wt(T) \leq 1 \tag{2.3}$$

They also proved that EDF is optimal among all algorithms for scheduling periodic task systems on uniprocessors. In other words, EDF correctly schedules any $\tau$ that satisfies (2.3).

### 2.1.2 Bin-packing Approaches

Several polynomial-time heuristics have been proposed for task partitioning based on bin-packing approaches [DD86, DL78]. We describe three of them below. While

```
NEXT-FIT()
begin
    k := 0;
    i := 0;
    while (i < N) and (k < M) do
        if task[i] fits on processor k then
            Assign task[i] to processor k;
            i := i + 1
        else
            k := k + 1
        fi
    od;
    if i < N then
        return failure
end
```

Figure 2.3: The next-fit partitioning heuristic.

describing these heuristics, we assume that there are $M$ processors numbered from 0 to $M-1$, and $N$ tasks numbered from 0 to $N-1$ that need to be scheduled. (The tasks are not assumed to be arranged in any specific order. We can obtain different variants of the these approaches by sorting the tasks in a specific order before applying the heuristics.) The schedulability test associated with the chosen uniprocessor scheduling algorithm can be used as an acceptance test to determine whether a task can "fit" on a processor. For instance, under EDF scheduling, a task will fit on a processor as long as the total utilization of all tasks assigned to that processor does not exceed unity (refer to (2.3)).

**Next Fit (NF):** In this approach, all the processors are considered in order, and we assign to each processor as many tasks as can fit on that processor. The procedure shown in Figure 2.3 describes this approach.

**First Fit (FF):** FF improves upon NF by also considering earlier processors during task assignment. Thus, each task is assigned to the first processor that can accept it, as described in the procedure shown in Figure 2.4.

**Best Fit (BF):** In this approach, each task is assigned to a processor that **(i)** can accept the task, and **(ii)** will have minimal remaining spare capacity after its addition. The procedure shown in Figure 2.5 describes this approach, with EDF as the uniprocessor scheduling algorithm and (2.3) as the acceptance test. (Here,

```
FIRST-FIT()
begin
    for i = 0 to N − 1 do
        k := 0;
        while (k < M) and (task[i] does not fit on processor k) do
            k := k + 1
        od;
        if (k < M) then
            Assign task[i] to processor k
        else
            return failure
        fi
    od
end
```

Figure 2.4: The first-fit partitioning heuristic.

```
BEST-FIT()
begin
    for i = 0 to N − 1 do
        min := −1;
        minSpare := 1;
        for k = 0 to M − 1 do
            currSpare := 1 − (W[k] + wt(task[i]));
            if (currSpare ≥ 0) and (currSpare < minSpare) then
                minSpare := currSpare;
                min := k
            fi
        od;
        if min ≥ 0 then
            Assign task[i] to the processor numbered min;
            W[min] := W[min] + wt(task[i])
        else
            return failure
        fi
    od
end
```

Figure 2.5: The best-fit partitioning heuristic.

$W[k]$ denotes the sum of the weights of the tasks assigned to processor $k$.)

Note that the time complexity of NF is $O(N)$ times the complexity of the uniprocessor schedulability test, while the time complexity of both FF and BF is $O(MN)$ times the complexity of the uniprocessor schedulability test.

A complementary approach to BF is *worst fit* (*WF*), in which each task is assigned to a processor that **(i)** can accept the task, and **(ii)** will have maximal remaining spare capacity after its addition. Though this approach does not try to maximize utilization, it results in a partitioning in which the workload is equally balanced among the different processors.

We now state some known results involving partitioning schemes that use RM and EDF for uniprocessor scheduling [BLOS95, DD86, DL78, LGDG00, OS95a, SVC98].

**Utilization bounds under EDF.** Surprisingly, the worst-case achievable utilization on $M$ processors for all of the above-mentioned heuristics (and also for an optimal partitioning algorithm) is at most $(M+1)/2$, even when an optimal uniprocessor scheduling algorithm such as EDF is used. In other words, there exist task systems with utilization slightly greater than $(M+1)/2$ that cannot be correctly scheduled by any partitioning approach. To see why, note that $M+1$ tasks, each with utilization $(1+\epsilon)/2$, cannot be partitioned on $M$ processors, regardless of the partitioning heuristic or the scheduling algorithm. As $\epsilon$ tends to 0, the total utilization of such a task system tends to $(M+1)/2$. Thus, we have the following theorem.

**Theorem 2.3.** *No partitioning-based scheduling algorithm can correctly schedule all task systems $\tau$ with $U(\tau) \leq B$ on $M$ processors for any $B > \frac{M+1}{2}$.*

Lopez *et al.* [LGDG00] showed that any task system with utilization at most $(M+1)/2$ can be correctly scheduled on $M$ processors if EDF is used as the uniprocessor scheduling algorithm with FF or BF as the partitioning heuristic.

Better utilization bounds can be obtained for EDF-scheduled systems if per-task utilizations are bounded. Let $\alpha$ denote the maximum utilization of any task in the system. Then any task can be assigned to a processor that has a spare capacity of at least $\alpha$. This implies that if a task system is not schedulable, then every processor must have spare capacity of less than $\alpha$. Hence, the total utilization of such a task system is more than $M(1-\alpha)+\alpha$. Equivalently, any task system with utilization at most $M-\alpha(M-1)$ is schedulable.[1] Lopez *et al.* [LGDG00] improved upon this by proving that EDF with FF (or BF) can correctly schedule any task system with utilization at most $(\beta M+1)/(\beta+1)$, where $\beta = \lfloor 1/\alpha \rfloor$.

[1] It is surprising to note that this schedulability test is identical to that for global EDF [GFB03].

**Theorem 2.4 (Lopez *et al.* [LGDG00]).** *If $U(\tau) \leq (\beta M + 1)/(\beta + 1)$, where $\beta = \lfloor 1/\alpha \rfloor$ and $\alpha$ satisfies $\alpha \geq wt(T)$ for all $T \in \tau$, then $\tau$ can be correctly scheduled by EDF with partitioning on $M$ processors.*

We now prove that the bound $(\beta M+1)/(\beta+1)$ is an improvement over the bound $M - \alpha(M-1)$. The following analysis shows that $(\beta M+1) > (\beta+1)(M-\alpha(M-1))$, which implies that $(\beta M+1)/(\beta+1) > (M-\alpha(M-1))$.

$$\begin{aligned}
& (\beta M + 1) - (\beta + 1)(M - \alpha(M-1)) \\
= \;& \left\lfloor \frac{1}{\alpha} \right\rfloor M + 1 - (\left\lfloor \frac{1}{\alpha} \right\rfloor + 1)(M - \alpha(M-1)) \\
= \;& \left\lfloor \frac{1}{\alpha} \right\rfloor M + 1 - \Big(\left\lfloor \frac{1}{\alpha} \right\rfloor M + M - \left\lfloor \frac{1}{\alpha} \right\rfloor \alpha(M-1) - \alpha(M-1)\Big) \\
= \;& 1 - M + \alpha \left\lfloor \frac{1}{\alpha} \right\rfloor (M-1) + \alpha(M-1) \\
= \;& \alpha(M-1)(\left\lfloor \frac{1}{\alpha} \right\rfloor + 1 - \frac{1}{\alpha}) \\
> \;& 0
\end{aligned}$$

(The last step follows the fact that $\lfloor x \rfloor > x - 1$.)

It is important to note that even though these utilization bounds are useful in understanding the worst-case behavior of partitioning with EDF, they are not typically used in practice. Better processor utilization can be usually obtained by directly using the above-described NF, FF, and BF implementations, unless efficiency considerations do not permit their use.

**Utilization bounds under RM.** The worst-case achievable utilization is much smaller for RM-scheduled systems since RM is not an optimal uniprocessor scheduling algorithm. Let $U_{RMFF}$ denotes the worst-case achievable utilization under RM with FF (RM-FF). Oh and Baker proved the following [OB98].

**Theorem 2.5 (Oh and Baker [OB98]).** $(\sqrt{2} - 1) \times M \leq U_{RMFF} \leq (M+1)/(1 + 2^{\frac{1}{M+1}})$.

Thus, task systems whose utilization do not exceed $(\sqrt{2}-1) \times M$ ($\approx 0.41 \times M$) are schedulable using RM-FF.

Several researchers have proposed partitioning heuristics that improve upon FF and BF. Oh and Son [OS95b] proposed an improved variant of the FF heuristic called First Fit Decreasing Utilization (FFDU). They showed that for RM-scheduled systems, the number of processors required by FFDU is at most 5/3 the optimal number of processors. (Dhall and Liu [DL78] had shown earlier that the number of processors required by FF and BF is at most twice the optimal number.)

Burchard *et al.* [BLOS95] presented new sufficient schedulability tests for RM-scheduled uniprocessor systems that perform better when task periods satisfy certain relationships. They also proposed new heuristics that try to assign tasks satisfying those relationships to the same processor, thus leading to better overall utilization. Lauzac *et al.* [LMM98] also proposed similar schedulability tests and heuristics, in which tasks are initially sorted in order of increasing periods. One disadvantage of these heuristics is the overhead of sorting, which may be unacceptable in an on-line setting.

### 2.1.3 Disadvantages of Partitioning

The use of partitioning is problematic in several classes of systems, as explained below.

**Dynamic task systems.** Implementing dynamic task systems systems by partitioning is problematic. To determine whether a new task can be allowed to join the system, the FF or BF heuristic needs to be executed, which takes at least $O(M)$ time. Note that if a new task must be admitted, then we may have to re-partition the entire system. Thus, determining whether the new set of tasks is feasible can be costly. Of course, the efficient schedulability test for EDF-FF given in Theorem 2.4 could be used, but its pessimism will likely result in much lower processor utilization.

**Resource and object sharing.** In most systems, tasks need to communicate with external devices and share resources. Thus, it may not be realistic to assume that all tasks are independent. To support non-independent tasks, schedulability tests are needed that take into account the use of shared resources. Such tests have been proposed for various synchronization schemes on uniprocessors, including the priority-inheritance protocol [SRL90], the priority-ceiling protocol [SRL90], the dynamic-priority-ceiling protocol [CL90], and EDF with dynamic deadline modification [Jef92].

Under partitioning, if all tasks that access a common resource can be assigned to the same processor, then the uniprocessor schemes cited above can be used directly. However, resource sharing across processors is often inevitable. For example, if the total utilization of all tasks that access a single resource is more than one, then, clearly, it is impossible for all of them to be assigned to the same processor. Also, even if the total utilization of such tasks is at most one, one of the tasks may access other shared resources. It might not be possible to assign all tasks accessing those resources to the same processor.

Adding resource constraints to partitioning heuristics is a non-trivial problem. Further, such constraints also make the uniprocessor schedulability test less tight, and hence, partitioning less effective. The multiprocessor priority-ceiling protocol (MPCP) was proposed for RM-scheduled systems by Rajkumar *et al.* [RSL88] as a means for synchronizing access to resources under partitioning. To the best of our knowledge, no multiprocessor synchronization protocols have been developed for partitioned systems with EDF (though it is probably not difficult to extend the MPCP for use in EDF-scheduled systems).

## 2.2 Proportionate Fair (Pfair) Scheduling

We now formally describe the fair scheduling concepts on which our work is based. In this section, we limit our attention to synchronous periodic task systems. (Recall that, in such systems, $T.\phi = 0$ for all tasks.) In Pfair scheduling, processor time is allocated in discrete time units, or *quanta*. We refer to the time interval $[t, t+1)$, where $t$ is a nonnegative integer, as *slot* $t$. (Hence, *time* $t$ refers to the beginning of slot $t$.) We further assume that all task parameters are expressed as integer multiples of the quantum size.[2]

The sequence of scheduling decisions over time defines a *schedule*. Formally, we represent a schedule $S$ as a mapping $S : \tau \times \mathcal{Z} \mapsto \{0, 1\}$, where $\tau$ is a set of periodic tasks and $\mathcal{Z}$ is the set of nonnegative integers. If $S(T, t) = 1$, then we say that *task* $T$ *is scheduled in slot* $t$. $S_t$ denotes the set of tasks scheduled in slot $t$. The statements $T \in S_t$ and $S(T, t) = 1$ are equivalent.

A task's weight defines the rate at which it is to be scheduled. In a perfectly fair (ideal) schedule, every task $T$ should receive a share of $wt(T) \cdot t$ time units over the interval $[0, t)$ (which implies that each job meets its deadline). In practice, this degree of fairness is impractical, as it requires the ability to preempt and switch tasks at arbitrarily small time scales. (Such idealized sharing is clearly not possible in a quantum-based schedule.) Instead, Pfair scheduling algorithms strive to "closely track" the allocation of processor time in the ideal schedule. This tracking is formalized in the

[2]This can always be ensured by either choosing a smaller quantum size or appropriately modifying task parameters, albeit with some schedulability loss.

notion of per-task *lag*, which is the difference between a task's allocation in the Pfair schedule and the allocation it would receive in an ideal schedule. Formally, the *lag of task $T$ at time $t$*, denoted $lag(T,t)$, is defined as follows.

$$lag(T,t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T,u). \tag{2.4}$$

(For brevity, we let the schedule be implicit and use $lag(T,t)$ instead of $lag(T,t,S)$. The schedule $S$ under consideration can be easily inferred from the context.) We can restate (2.4) as follows.

$$lag(T,t+1) = \begin{cases} lag(T,t) + wt(T), & \text{if } T \notin S_t \\ lag(T,t) + wt(T) - 1, & \text{if } T \in S_t. \end{cases} \tag{2.5}$$

Task $T$ is said to be *over-allocated* or *ahead* at time $t$, if $lag(T,t) < 0$, *i.e.*, the actual processor time received by $T$ over $[0,t)$ is more than its ideal share over $[0,t)$. Analogously, task $T$ is said to be *under-allocated* or *behind* at time $t$ if $lag(T,t) > 0$. If $lag(T,t) = 0$, then $T$ is *punctual*, *i.e.*, it is neither ahead nor behind.

A schedule is defined to be *proportionate fair* (*Pfair*) if and only if

$$(\forall T, t :: -1 < lag(T,t) < 1). \tag{2.6}$$

Informally, the allocation error for each task is always less than one quantum. From this, we have the following theorem.

**Theorem 2.6 (Baruah *et al.* [BCPV96]).** *In a Pfair schedule for a synchronous periodic task system, each task $T$ receives a share of either $\lfloor wt(T) \cdot t \rfloor$ or $\lceil wt(T) \cdot t \rceil$ time units over the interval $[0,t)$.*

*Proof.* By (2.4) and (2.6), we have $wt(T) \cdot t - \sum_{u=0}^{t-1} S(T,u) < 1$. Therefore, $\sum_{u=0}^{t-1} S(T,u) > wt(T) \cdot t - 1$. Since $S(T,u)$ is an integer, it follows that $\sum_{u=0}^{t-1} S(T,u) \geq \lfloor wt(T) \cdot t \rfloor$. Thus, $T$ receives a share of at least $\lfloor wt(T) \cdot t \rfloor$ time units over the interval $[0,t)$.

Again, by (2.4) and (2.6), we have $wt(T) \cdot t - \sum_{u=0}^{t-1} S(T,u) > -1$. Therefore, $\sum_{u=0}^{t-1} S(T,u) < wt(T) \cdot t + 1$. Hence, $\sum_{u=0}^{t-1} S(T,u) \leq \lceil wt(T) \cdot t \rceil$. Thus, $T$ receives a share of at most $\lceil wt(T) \cdot t \rceil$ time units over the interval $[0,t)$.

Because $\lceil wt(T) \cdot t \rceil - \lfloor wt(T) \cdot t \rfloor$ is at most 1, it follows that $T$ receives a share of either $\lfloor wt(T) \cdot t \rfloor$ or $\lceil wt(T) \cdot t \rceil$ time units over the interval $[0,t)$. ■

It is straightforward to show that in any Pfair schedule, each job of a periodic task meets its deadline. Note that, if $t = k \cdot T.p$, then $\lfloor wt(T) \cdot t \rfloor$ and $\lceil wt(T) \cdot t \rceil$ both reduce to $k \cdot T.e$. Therefore, by Theorem 2.6, the share received by a task $T$ over the interval $[0, k \cdot T.p)$ is exactly $k \cdot T.e$. Thus, $T$ receives exactly $T.e$ units of processor time over $[(k-1) \cdot T.p, k \cdot T.p)$. This yields the following theorem.

**Theorem 2.7 (Baruah *et al.* [BCPV96]).** *Each job of a periodic task in a synchronous periodic task system meets its deadline in a Pfair schedule.*

**Subtasks and Pfair windows.** We refer to each quantum of execution of a task as a *subtask*; thus, the execution of each task $T$ corresponds to the execution of an infinite sequence of subtasks. We denote the $i^{th}$ subtask of task $T$ as $T_i$, where $i \geq 1$. Thus, the $k^{th}$ job ($k \geq 0$) of task $T$ consists of the subtasks $T_{ke+1}, \ldots, T_{(k+1)e}$, where $e = T.e$. Subtask $T_{i+1}$ is called the *successor* subtask of $T_i$ and $T_i$ is called the *predecessor* subtask of $T_{i+1}$.

By Theorem 2.6, each task receives either $\lfloor wt(T) \cdot t \rfloor$ or $\lceil wt(T) \cdot t \rceil$ units of processor time in a Pfair schedule. To enforce this in any quantum-based schedule, each subtask $T_i$ has an associated *pseudo-release*, denoted $r(T_i)$, and *pseudo-deadline*, denoted $d(T_i)$. $r(T_i)$ is the first slot into which $T_i$ potentially could be scheduled, and $d(T_i)$ is the time before which it must finish execution. We now derive simple formulae for $r(T_i)$ and $d(T_i)$.

Note that if $lag(T, t) \leq -wt(T)$ and $T$ is scheduled in slot $t$, then by (2.5), $lag(T, t+1) \leq -wt(T) + wt(T) - 1$, *i.e.*, $lag(T, t+1) \leq -1$. This violates the lag bounds in (2.6). Hence, $T$ is eligible for execution at time $t$ if and only if $lag(T, t) > -wt(T)$. Thus, $r(T_i)$ corresponds to the smallest $t$ such that $lag(T, t) > -wt(T)$ after the first $i - 1$ subtasks of $T$ have completed execution. This implies that $r(T_i)$ is the smallest $t$ such that $wt(T) \cdot t - (i-1) > -wt(T)$, *i.e.*, $t > \dfrac{i-1}{wt(T)} - 1$. Thus, we have the following.

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{2.7}$$

Recall that slot $t$ refers to the interval $[t, t+1)$. Thus, by the definition of $d(T_i)$, if $T_i$ is not scheduled in slot $d(T_i) - 1$, then $T$ will violate its lag bounds at time $d(T_i)$. Therefore, $d(T_i) - 1$ is the largest $t$ such that $wt(T) \cdot t - (i-1) < 1$ (by (2.4)), *i.e.*, $t < \dfrac{i}{wt(T)}$. Therefore, $d(T_i)$ is the largest $u$ such that $u < \dfrac{i}{wt(T)} + 1$, which implies

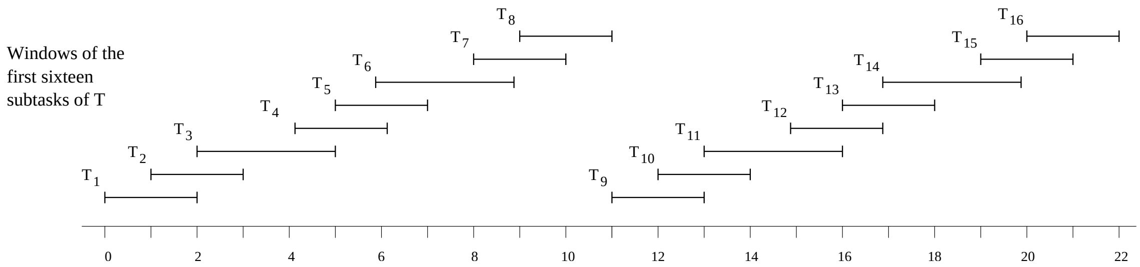



Figure 2.6: The Pfair windows of the first two jobs (or sixteen subtasks) of a task $T$ with weight 8/11 in a Pfair-scheduled system. During each job of $T$, each of the eight units of computation must be allocated processor time during its window, or else a lag-bound violation will result.

that $u \leq \left\lceil \frac{i}{wt(T)} \right\rceil$. Thus, we have the following.

$$d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \tag{2.8}$$

For brevity, we often refer to pseudo-releases and pseudo-deadlines as simply releases and deadlines, respectively. The interval $[r(T_i), d(T_i))$ is called the *window* of subtask $T_i$ and is denoted by $w(T_i)$. The *length* of window $w(T_i)$, denoted $|w(T_i)|$, is defined as $d(T_i) - r(T_i)$. Thus, by (2.7) and (2.8), we have the following.

$$|w(T_i)| = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{2.9}$$

We refer to a window of length $n$ as an *n-window*. From (2.7)–(2.9), it is easy to see that the sequence of windows of a task $T$ and $U$ are identical if $wt(T) = wt(U)$ (even if $T.e \neq U.e$). Thus, under Pfair scheduling, both tasks are treated in exactly the same manner.

As an example, consider a task $T$ with weight $wt(T) = 8/11$. Each job of this task consists of eight subtasks, and hence each period is eight (overlapping) windows. Using Equations (2.7) and (2.8), it is easy to show that the windows within each job of $T$ are as depicted in Figure 2.6. As mentioned above, a task with weight 16/22 has the same sequence of windows.

### 2.2.1 Feasibility

Baruah *et al.* proved the following feasibility condition for the Pfair scheduling of synchronous periodic tasks [BCPV96].

$$\sum_{T \in \tau} \frac{T.e}{T.p} \leq M \tag{2.10}$$

This was shown by means of a network flow construction. Let $L$ denote the least common multiple of $\{T.p \mid T \in \tau\}$. By restricting attention to subtasks that are released within the first hyperperiod of $\tau$ (*i.e.*, in the interval $[0, L)$), the sufficiency of (2.10) can be established by constructing a flow graph with integral edge capacities and by then applying the Ford-Fulkerson result [FF62] to prove the existence of an integer-valued maximum flow for that graph. This integral flow defines a correct schedule over $[0, L)$. (Later, in Chapter 4, we use this same technique to obtain a feasibility condition for task systems belonging to a more general task model. Therefore, we do not describe this construction in greater detail here.)

**Theorem 2.8 (Baruah *et al.* [BCPV96]).** *A synchronous, periodic task system $\tau$ has a Pfair schedule on $M$ processors if and only if $\sum_{T \in \tau} \frac{T.e}{T.p} \leq M$.*

### 2.2.2 The PF Pfair Algorithm

PF was the first Pfair scheduling algorithm that was shown to be optimal on multiprocessors [BCPV96]. PF prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis and uses several tie-breaking rules when multiple subtasks have the same deadline. The first tie-breaking rule involves a parameter called the *successor bit*, which is defined as follows.

$$b(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i}{wt(T)} \right\rfloor \tag{2.11}$$

Thus, $b(T_i)$ is either 0 or 1. Informally, $b(T_i)$ denotes the number of slots by which $T_{i+1}$'s window overlaps $T_i$'s window (see (2.7) and (2.8)). For example, in Figure 2.6, $b(T_i) = 1$ for $1 \leq i \leq 7$ and $b(T_8) = 0$.

If $T.e$ divides $i$, then $(i \cdot T.p)/T.e$ is an integer, *i.e.*, $\left\lceil \frac{i}{wt(T)} \right\rceil = \left\lfloor \frac{i}{wt(T)} \right\rfloor$. Thus, we have the following property.

**(B)** If $T_i$ is the last subtask of a job, then $b(T_i)$ is zero.

Under the PF algorithm, subtasks are prioritized as follows: at time $t$, if subtasks $T_i$ and $U_j$ are both ready to execute, then $T_i$'s priority is at least that of $U_j$, denoted $T_i \preceq U_j$, if one of the following rules is satisfied.

**(i)** $d(T_i) < d(U_j)$.

**(ii)** $d(T_i) = d(U_j)$ and $b(T_i) > b(U_j)$.

**(iii)** $d(T_i) = d(U_j)$, $b(T_i) = b(U_j) = 1$, and $T_{i+1} \preceq U_{j+1}$.

If neither subtask has priority over the other, then the tie can be broken arbitrarily. Given the PF priority definition, the description of the PF algorithm is simple: at the start of each slot, the $M$ highest priority subtasks (if that many eligible subtasks exist) are selected to execute in that slot.

As shown in Rule (ii), when comparing two subtasks with equal deadlines, PF favors a subtask $T_i$ with $b(T_i) = 1$, *i.e.*, if its window overlaps that of its successor. The intuition behind this rule is that executing $T_i$ early reduces its chances of getting scheduled in its last slot. The latter possibility effectively leads to a shortening of $w(T_{i+1})$, and imposes more constraints on the scheduling of $T_{i+1}$. If two subtasks have equal deadlines and successor bits of 1, then according to Rule (iii), their successor subtasks are recursively checked. This recursion will halt within $\mathbf{min}(T.e, U.e)$ steps, because the last subtask of each job has a successor bit of 0 (by (B)).

We now briefly sketch the proof of Baruah *et al.* that PF is optimal.

**Theorem 2.9 (Baruah *et al.* [BCPV96]).** PF *is optimal for scheduling synchronous periodic tasks on multiprocessors.*

*Proof sketch.* The optimality proof for PF proceeds by inducting over the interval $(0, L]$, where $L$ is the least common multiple of $\{T.p \mid T \in \tau\}$. The crux of the argument is to show that, if there exists a Pfair schedule $S$ such that all decisions in $S$ before slot $t$ are in accordance with PF priorities, then there exists a Pfair schedule $S'$ such that all the scheduling decisions in $S'$ before slot $t+1$ are in accordance with PF priorities. To prove the existence of $S'$, the scheduling decisions in slot $t$ of $S$ are systematically changed so that they respect the PF priority rules, while maintaining the correctness of the schedule. We briefly summarize the swapping arguments used to transform $S$.

Let $\prec$ be an irreflexive total order that is consistent with the $\preceq$ relation in the PF priority definition, *i.e.*, $\prec$ is obtained by arbitrarily breaking any ties left by $\preceq$. Suppose that $T_i$ and $U_j$ are both eligible to execute in slot $t$ and $T_i \prec U_j$. Furthermore,

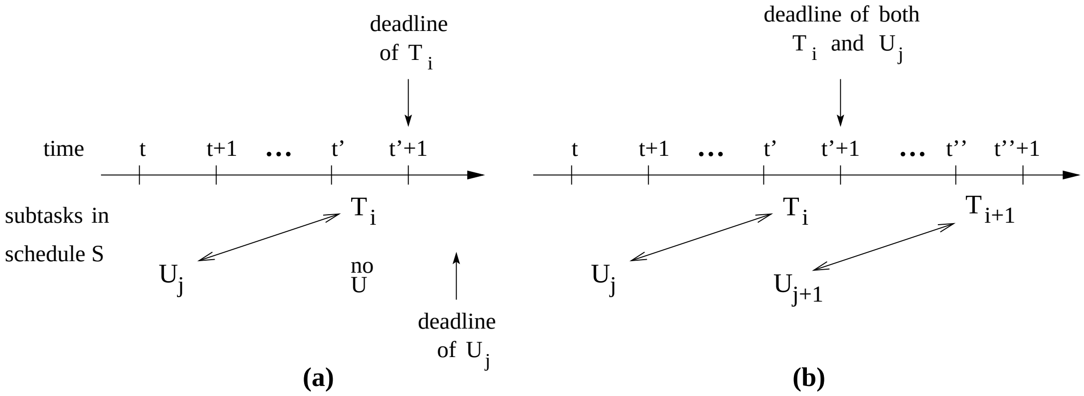



Figure 2.7: Optimality proof for PF. A double arrow indicates two subtasks that are to be swapped. In (a), "no $U$" means no subtask of $U$ is scheduled in slot $t'$.

suppose that, contrary to the PF priority rules, $U_j$ is scheduled in slot $t$ in $S$, while $T_i$ is scheduled in a later slot $t'$ in $S$. Since $T_i \preceq U_j$, there are three possibilities.

- $T_i \prec U_j$ **by Rule (i).** Because $T_i$'s deadline is less than $U_j$'s deadline, and because the windows of consecutive subtasks overlap by at most one slot, $U_{j+1}$ is scheduled at a later slot than $T_i$. Therefore, $T_i$ and $U_j$ can be directly swapped to get the desired schedule, as shown in Figure 2.7(a).

- $T_i \prec U_j$ **by Rule (ii).** By Rule (ii), $b(U_j) = 0$, which implies that $U_{j+1}$'s window does not overlap that of $U_j$. Hence, $T_i$ and $U_j$ can be directly swapped without affecting the scheduling of $U_{j+1}$, as shown in Figure 2.7(a).

- $T_i \prec U_j$ **by Rule (iii).** (This is the difficult case to consider. As we shall see later in Section 3.4, a major portion of the optimality proof of PD$^2$ deals with this case.) In this case, it may not be possible to directly swap $T_i$ and $U_j$ because $U_{j+1}$ may be scheduled in the same slot as $T_i$ (*i.e.*, swapping $T_i$ and $U_j$ would result in $U_j$ and $U_{j+1}$ being scheduled in the same slot). If $U_{j+1}$ is indeed scheduled in slot $t'$, then it is necessary to first swap $T_{i+1}$ and $U_{j+1}$, as shown in Figure 2.7(b), which may in turn necessitate the swapping of later subtasks. By Rule (iii), $T_{i+1} \preceq U_{j+1}$, and hence we can inductively repeat the above procedure to swap $T_{i+1}$ and $U_{j+1}$.

This completes the proof of the theorem. ■

Though optimal, PF is inefficient due to the recursion in Rule (iii). In particular, given two subtasks $T_i$ and $U_j$, determining which job has higher priority takes

$O(\log(\mathbf{min}(T.p, U.p)))$ time. Baruah, Gehrke, and Plaxton [BGP95] presented a more efficient algorithm called PD in which Rule (iii) is replaced by three additional rules, each of which involves only an $O(1)$-time calculation. As mentioned earlier, our new PD$^2$ algorithm is the most efficient among all known optimal Pfair scheduling algorithms. In particular, PD$^2$ was obtained from the PD algorithm by replacing Rule (iii) with only a single rule that takes $O(1)$ time. Since the priority definitions of both algorithms are very similar, we discuss PD after presenting PD$^2$ (in the next chapter).

### 2.2.3 Related Work on Fair Scheduling

Much work has been devoted to uniprocessor fair scheduling algorithms based on *generalized processor sharing* (*GPS*). The concept of GPS is similar to the concept of an ideal scheduler used in the definition of Pfairness, except that, under GPS, excess spare capacity is distributed fairly among all the tasks in proportion to their weights. Thus, a task $T$ would receive a share according to a weight of $\frac{wt(T)}{\sum_{U \in A} wt(U)}$, where $A$ denotes the set of active tasks at any instant. As with the ideal scheduler considered earlier, GPS is also impossible to achieve in practice. Several researchers have designed practical algorithms that approximate GPS to various degrees. We review some of this research in this section, and also discuss how it differs from Pfair scheduling.

Demers *et al.* [DKS89] presented the GPS-based *weighted fair queueing* (*WFQ*) algorithm for scheduling packet transmissions in routers. They also presented the concept of *virtual time* as a means to efficiently keep track of the shares of each flow as new connections are established and old ones are disconnected. Under WFQ, a virtual deadline is assigned to each packet based on its size and the share of its associated connection. At any instant, the packet with earliest deadline is selected to be transmitted. Parekh and Gallager [PG93, PG94] proved that the WFQ algorithm guarantees constant lag bounds. In particular, no packet is serviced $P_{max}$ later than it would have been serviced in the fluid-flow system, where $P_{max}$ is the time required to transmit a packet of maximum size. This initial work of fair packet scheduling has been extended in several ways. Several algorithms that improve the efficiency of WFQ at the expense of higher lag bounds have been proposed [Gol94, GVC96]. Conversely, the worst-case fair WFQ (WF$^2$Q) algorithm provides better lag bounds with a small reduction in efficiency [BZ96]. Hierarchical schedulers have also been proposed, thus allowing two or more connections to be combined and scheduled as a single entity [BZ97, SZN97].

Stoica *et al.* [SAWJ$^+$96] extended this work to quantum-based scheduling in operating systems by developing the concept of *proportional share* scheduling. They also presented the quantum-based earliest-eligible-virtual-deadline-first (EEVDF) algorithm and proved lag bounds similar to (2.6), which in turn provide bounds on timeliness. Fair schedulers are useful in general-purpose operating systems as they provide *temporal isolation*, *i.e.*, a misbehaving task does not adversely affect the scheduling of other tasks in the system. Jeffay *et al.* [JSMA98] showed that the problem of receive livelock[3] can be ameliorated by the proportional-share scheduling of kernel activities.

Unfortunately, all of the above-described research on GPS-based algorithms is applicable only to uniprocessor systems. Though other researchers have recently explored the use of fair scheduling algorithms on multiprocessors, most of these results have been empirical. In particular, Chandra, Adler, and Shenoy investigated the use of fair scheduling algorithms (based on the $\mathrm{PD}^2$ algorithm developed by us) in the Linux operating system [CAS01]. The goal of their work was to determine the efficacy of using fair scheduling to provide quality-of-service guarantees to multimedia applications. Consequently, no formal analysis of their approach was presented. Despite this, their experimental results convincingly demonstrate the utility of fair scheduling in multiprocessor systems.

Chandra *et al.* [CAGS00] extended the concept of GPS to multiprocessors to obtain *generalized multiprocessor scheduling* (*GMS*). Note that the share calculation used in GPS (described earlier) may not always work on a multiprocessor because it can result in a share larger than one. GMS resolves this by capping the maximum share of any task to one. Chandra *et al.* presented an algorithm (based on the uniprocessor WFQ algorithm) to approximate GMS, and demonstrated its effectiveness through an implementation on a Linux platform. However, they too present no formal analysis of their scheduling algorithms.

## 2.3 Summary

Though Pfair scheduling algorithms hold much promise, prior to our work, research on this topic was limited in several ways. First, only synchronous periodic task systems were considered. Second, Pfair scheduling (as originally defined) is necessarily not work-conserving when used to schedule periodic tasks. Third, results prior to our work do not apply to scheduling of dynamic task systems on multiprocessors. Fourth, no approaches

[3]Receive livelock refers to a condition in which the processor spends all of its time processing interrupts (that are generated due to packets arriving over a network) without doing any useful work.

for handling soft real-time tasks and aperiodic tasks were known. We address these concerns in this dissertation. We start by describing the PD$^2$ Pfair algorithm, and the concept of ERfair scheduling in the next chapter. In addition, we prove that PD$^2$ is an optimal algorithm for scheduling periodic task systems on multiprocessors and present results of experiments that compare PD$^2$ and the EDF-FF partitioning algorithm.

# Chapter 3

# The PD$^2$ Scheduling Algorithm*

In this chapter, we present the PD$^2$ algorithm, and also introduce the concept of ERfair scheduling. We also prove that PD$^2$ is optimal for scheduling periodic task systems.

## 3.1 The PD$^2$ Priority Definition

We first describe the PD$^2$ priority definition and then discuss how it differs from the PD priority definition [BGP95]. Both PD$^2$ and PD classify tasks into two categories depending on their weight: a task $T$ is *light* if $wt(T) < 1/2$, and *heavy* otherwise.

In the PD$^2$ priority definition, the recursive Rule (iii) of PF (refer to Section 2.2.2) is replaced by a simple comparison of the "group deadlines" of competing subtasks. Group deadlines are only important when heavy tasks of weight less than one exist, *i.e.*, when $1/2 \le wt(T) < 1$ holds for some task $T$. (As shown later in Section 3.4.1, a task $T$ has windows of length two if and only if $1/2 \le wt(T) < 1$.) If a task does not satisfy this criterion, then its group deadline is defined to be 0.

---

*The results presented in this chapter have been published in the following papers.

[AS04] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer Systems and Sciences*, 2004. To appear. (Most of the results in this paper were presented in preliminary form at the 12th and 13th Euromicro Conferences on Real-time Systems [AS00a, AS01].)

[SHAB03] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of the 11th International Workshop on Parallel and Distributed Real-Time Systems*, April 2003. (On CD-ROM.)

**The group deadline.** To motivate the definition of the group deadline, consider a sequence $T_i, \ldots, T_j$ of subtasks such that $b(T_k) = 1 \ \wedge \ |w(T_{k+1})| = 2$ for all $i \le k < j$. Note that scheduling $T_i$ in its last slot forces the other subtasks in this sequence to be scheduled in their last slots. For example, in Figure 2.6, scheduling $T_3$ in slot 4 forces $T_4$ and $T_5$ to be scheduled in slots 5 and 6, respectively. The group deadline of a subtask $T_i$, denoted $D(T_i)$, is the earliest time by which such a "cascade" must end. Formally, it is the earliest time $t$, where $t \ge d(T_i)$, such that either $(t = d(T_k) \ \wedge \ b(T_k) = 0)$ or $(t + 1 = d(T_k) \ \wedge \ |w(T_k)| = 3)$ for some subtask $T_k$. For example, in Figure 2.6, $D(T_3) = d(T_6) - 1 = 8$ and $D(T_7) = d(T_8) = 11$. PD$^2$ favors subtasks with later group deadlines because *not* scheduling them can lead to longer cascades of scheduling decisions, thus placing more constraints on the future schedule. We now derive a formula for determining a subtask's group deadline.

Consider a Pfair schedule $S$ for a single task $T$ in which each subtask of $T$ is scheduled in the first slot of its window. Then, the slots that remain empty in $S$ exactly correspond to the group deadlines of $T$ (refer to Figure 3.1(a)). Thus, the number of group deadlines in every period of $T$ is $T.p - T.e$.

We now show that the group deadlines of $T$ correspond to the subtask deadlines of a task $U$ such that $U.e = T.p - T.e$ and $U.p = T.p$, which implies that $wt(U) = 1 - wt(T)$. Consider a schedule $S'$ for task $U$ obtained from $S$ as follows: $U$ is scheduled in slot $t$ in $S'$ if and only if $T$ is *not* scheduled in slot $t$ in $S$. Figure 3.1 illustrates $S$ and $S'$ for a task $T$ of weight $8/11$. Consider $T$'s *lag* in schedule $S$, and $U$'s *lag* in schedule $S'$. (Hence, $lag(T, t)$ refers to schedule $S$, and $lag(U, t)$ refers to schedule $S'$.) By (2.4), we have $lag(U, t) = wt(U) \cdot t - \sum_{u \in [0,t)} S'(U, u)$, which is equivalent to $(1 - wt(T)) \cdot t - (t - \sum_{u \in [0,t)} S(T, u))$. Simplifying this, we obtain $lag(U, t) = \sum_{u \in [0,t)} S(T, u) - wt(T) \cdot t$. Therefore, by (2.4), $lag(U, t) = -lag(T, t)$. Thus, if $T$ satisfies (2.6), then $U$ also satisfies (2.6), and hence, $S'$ is a Pfair schedule.

Note that for any $t$ such that $t = D(T_i) - 1$, $T$ is not scheduled in slot $t$ in $S$. Since each subtask is scheduled in the first slot of its window (*i.e.*, as early as possible), this implies that $T$ is not eligible for execution at time $t$. Therefore, $lag(T, t) \le -wt(T)$, which implies that $lag(U, t) \ge wt(T)$. If $U$ is not scheduled in slot $t$, then, by (2.5), $lag(U, t + 1) \ge wt(T) + (1 - wt(T))$, *i.e.*, $lag(U, t + 1) \ge 1$. This implies that $t + 1$ $(= D(T_i))$ is a pseudo-deadline of $U$. Thus, we have the following theorem.

**Theorem 3.1.** *The group deadlines of a task $T$ correspond to the subtask deadlines of a task $U$ such that $U.e = T.p - T.e$ and $U.p = T.p$, i.e., $wt(U) = 1 - wt(T)$.*

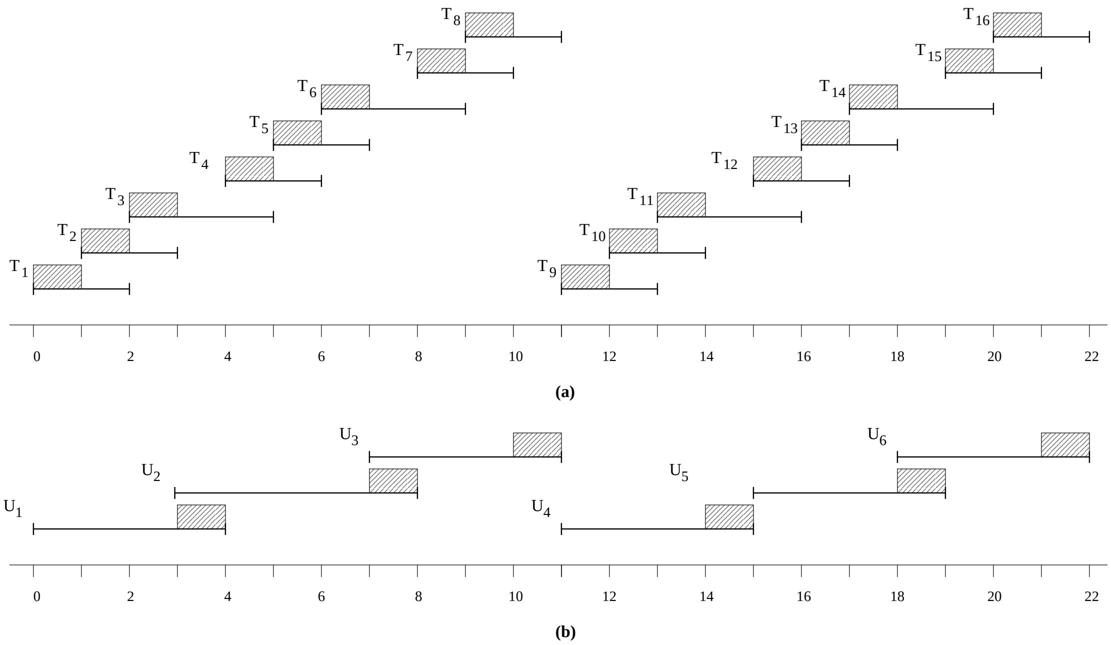



Figure 3.1: **(a)** A schedule $S$ for task $T$ of weight 8/11 such that each subtask is scheduled in the first slot of its window. The group deadlines of $T$ over $(0, 22]$ are at times 4, 8, 11, 15, 19, and 22, which exactly correspond to slots that remain empty in $S$. **(b)** Schedule $S'$ for task $U$ of weight 3/11. $U$ is scheduled in slot $t$ if and only if $T$ is not scheduled in $t$ in $S$. $T$'s group deadlines correspond to the pseudo-deadlines of $U$.

Thus, $D(T_i)$ can be obtained by using (2.8) to calculate the subtask deadlines of a task $U$ with $wt(U) = 1 - wt(T)$. For any subtask $T_i$, $D(T_i)$ corresponds to $d(U_k)$, where $k$ is the smallest $j$ such that $d(U_j) \geq d(T_i)$. For example, in Figure 3.1, $D(T_3) = d(U_2) = 8$. Thus, the scheduler can calculate the current group deadline of $T$ by keeping track of the subtask indices of $U$. Below, we derive another formula that provides us with a more direct way to calculate group deadlines. In this derivation, we consider the following two cases separately: $d(U_k) = d(T_i)$ and $d(U_k) > d(T_i)$.

If $d(U_k) = d(T_i)$, then $D(T_i) = d(T_i)$. By definition of a group deadline, this implies that $b(T_i) = 0$. We now show that $b(U_k) = 0$. By (2.11), it follows that $\frac{i}{wt(T)}$ in an integer. By (2.4), at time $t = \frac{i}{wt(T)}$ (which equals $d(T_i)$ by (2.8)), $lag(T, t)$ must be an integer. By (2.6), this implies that $lag(T, t) = 0$. Hence, $lag(U, t) = -lag(T, t) = 0$. Thus, by (2.4), this implies that $wt(U) \cdot t$ must be an integer. Therefore, $t = \frac{j}{wt(U)}$ for some $j$. Because $t = d(T_i) = d(U_k)$, it follows that $d(U_k) = \frac{j}{wt(U)}$. By

(2.8), and because $\frac{j}{wt(U)}$ is an integer, we have $k = j$. Thus, by (2.11), $b(U_k) = 0$. Therefore, by (2.11), $\left\lceil \frac{k}{1 - wt(T)} \right\rceil = \frac{k}{1 - wt(T)}$. By (2.8), $d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil$, and $d(U_k) = \left\lceil \frac{k}{1 - wt(T)} \right\rceil$. Hence, $\frac{k}{1 - wt(T)} = \left\lceil \frac{i}{wt(T)} \right\rceil$, *i.e.*, $k = \left\lceil \frac{i}{wt(T)} \right\rceil (1 - wt(T))$. Note that the right-hand-side of this equation must be an integer, and hence, we have $k = \left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil (1 - wt(T)) \right\rceil$.

If $d(U_k) > d(T_i)$, then by (2.8), we have $\left\lceil \frac{k}{1 - wt(T)} \right\rceil > \left\lceil \frac{i}{wt(T)} \right\rceil$. This implies that $\frac{k}{1 - wt(T)} > \left\lceil \frac{i}{wt(T)} \right\rceil$. Therefore, $k > \left\lceil \frac{i}{wt(T)} \right\rceil (1 - wt(T))$, implying that $k \geq \left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil (1 - wt(T)) \right\rceil$. Since $k$ is the smallest such index, it follows that $k = \left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil (1 - wt(T)) \right\rceil$.

Thus, under both cases, because $D(T_i) = d(U_k)$, by (2.8), we obtain the following formula.

$$D(T_i) = \left\lceil \frac{\left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil \times (1 - wt(T)) \right\rceil}{1 - wt(T)} \right\rceil \tag{3.1}$$

**Priority Definition.** We can now state the PD$^2$ priority definition. (For the sake of completeness, we repeat Rules (i) and (ii) from Section 2.2.2.) Under PD$^2$, subtask $T_i$'s priority is at least that of subtask $U_j$, denoted $T_i \preceq U_j$, if one of the following rules is satisfied.

**(i)** $d(T_i) < d(U_j)$.

**(ii)** $d(T_i) = d(U_j)$ and $b(T_i) > b(U_j)$.

**(iii)** $d(T_i) = d(U_j)$, $b(T_i) = b(U_j) = 1$, and $D(T_i) \geq D(U_j)$.

Any ties not resolved by these three rules can be broken arbitrarily. Thus, according to this definition, $T_i$ has higher priority than $U_j$ if it has an earlier pseudo-deadline. If $T_i$ and $U_j$ have equal pseudo-deadlines, but $b(T_i) = 1$ and $b(U_j) = 0$, then the tie is broken in favor of $T_i$. If $T_i$ and $U_j$ have equal pseudo-deadlines and successor bits of one, then their group deadlines are inspected to break the tie. If one is heavy and the other light, then the tie is broken in favor of the heavy task. If both are heavy and their group deadlines differ, then the tie is broken in favor of the one with the larger group deadline. If both are heavy and have equal group deadlines, then the tie can be broken arbitrarily. (Recall that $D(T_i) = 0$ if $T$ is light. Thus, in a light-only task system, PD$^2$ uses only the first two rules to determine subtask priorities.)

**The PD algorithm.** The priority definition used in the PD algorithm [BGP95] has two tie-break parameters (and rules) in addition to those used by PD$^2$. The first is a bit that distinguishes between the two different types of group deadline. For example, in Figure 3.1, $D(T_1)$ is a type-1 group deadline because it corresponds to the middle slot of a 3-window, while $D(T_6)$ is a type-0 group deadline. (Effectively, the value of this bit is the value of $b(U_k)$, where $U_k$ is the subtask of $U$ that defines the group deadline of $T$.) The second parameter $T.x$ is defined as follows: for a light task $T.x = \left\lfloor \frac{1}{wt(T)} \right\rfloor$, and for a heavy task $T.x = \left\lfloor \frac{1}{1 - wt(T)} \right\rfloor$. PD was proved optimal by Baruah *et al.* [BGP95] by a simulation argument that shows that PD "closely tracks" the behavior of the PF algorithm; the optimality of PF was then used to infer the optimality of PD [BGP95]. The optimality of PD$^2$ (as proved later) shows that the two additional tie-breaking rules of PD are not needed.

**Implementation.** The PD$^2$ algorithm is nearly identical to the algorithm given for PD in [BGP95], except that a different priority definition is used. As shown in [BGP95], PD$^2$ can be implemented in $O(\mathbf{min}(N, M \log N))$ time, where $M$ is the number of processors and $N$ is the total number of tasks.

An $O(N)$ implementation can be obtained simply by using a comparison-based selection algorithm that runs in $O(N)$ time [BFP$^+$73]. In particular, if the number of tasks that are eligible is greater than $M$, then the subtask with the $M^{th}$ highest priority according to PD$^2$ can be obtained in $O(N)$ time using the above-mentioned selection algorithm. The rest of the tasks can be partitioned using this subtask's priority and thus, the remaining $(M - 1)$ tasks can also be chosen in $O(N)$ time.

Figure 3.2 describes a heap-based implementation that runs in $O(M \log N)$ time. A priority-ordered *ready queue* is used to store eligible subtasks. In addition, there are a number of priority-ordered *release queues* associated with future time slots. At the beginning of each time slot, the $M$ highest-priority subtasks in the ready queue (if that many subtasks are eligible) are selected for execution. If $T_i$ is one of the selected subtasks, then $T_{i+1}$ is inserted into the release queue associated with time $t'$, where $t'$ is the time at which $T_{i+1}$ becomes eligible (lines 8 and 21), *i.e.*, $t' = \mathbf{max}(r(T_{i+1}), t + 1)$. At the beginning of each time slot, the release queue for that slot is merged with the

```
ALGORITHM PD²
0:  begin
1:      H := BuildHeap(τ);
2:      t := 0;
3:      when scheduling slot t do
4:          repeat
5:              T:= ExtractMin(H);
6:              "Schedule task T in slot t";
7:              t' := "the earliest future time at which task T will be eligible again";
8:              Requeue(T, t')
9:          until "M tasks have been scheduled in slot t";
10:         if "Heap H_{t+1} exists" then
11:             H := Union(H, H_{t+1})
12:         fi;
13:         t := t + 1
14:     od
15: end
```

```
Requeue(T, t)
16: begin
17:     if "Heap H_t does not exist" then
18:         H_t := MakeHeap()
19:     fi;
20:     "Determine T's priority at time t";
21:     Insert(H_t, T)
22: end
```

Figure 3.2: An $O(M \log N)$ implementation of PD$^2$

ready queue (line 11). An additional search structure based on red-black trees can be used in order to efficiently access the release queues [BGP95].

The above-described procedure can be implemented in $O(M \log N)$ time using the *binomial-heap* data structure [Vui78]; the primary reason for using binomial heaps is that two such heaps can be merged in $O(\log n)$ time, where $n$ is the total number of items in both heaps.

## 3.2 Minimality of the PD$^2$ Priority Definition

Before proving the optimality of PD$^2$, we consider other scheduling algorithms that determine subtask priorities using fewer or more-efficient tie-breaking rules.

According to the $\mathrm{PD}^2$ priority definition, each task $T$ is effectively prioritized at time $t$ by the triple $(d(T_i), b(T_i), D(T_i))$, where $T_i$ is the subtask of $T$ eligible at time $t$. In this section, we present a collection of counterexamples that show that this priority definition cannot be substantially simplified.

In each proof in this section, an example task system is considered that fully utilizes a system of $M$ processors for some $M$. Each such task system consists of a set $A$ of tasks of one weight and a set $B$ of tasks of another weight. We show that if this task system is scheduled with the newly-proposed priority definition, then a time slot is reached at which fewer than $M$ tasks are scheduled. Since the task system fully utilizes the $M$ processors, this implies that a deadline is missed at some future time. In the proof of Theorem 3.2, we explain the resulting schedule in detail. The subsequent proofs in this section are sketched more briefly. We begin by considering the $b$-bit.

**Theorem 3.2.** *If the* $\mathrm{PD}^2$ *priority definition is changed by eliminating* $b$ (*i.e., Rule* (ii))*, then there exists a feasible task system that is not correctly scheduled.*

*Proof.* Consider a task system consisting of a set $A$ of eight light tasks with weight 1/3 and a set $B$ of three light tasks with weight 4/9. Because total utilization is four, Expression (2.10) implies that the system is feasible on four processors. Consider the schedule shown in Figure 3.3(a).

As seen in Figure 3.3(a), each job of a task with weight 1/3 consists of one three-slot window. Each job of a task with weight 4/9 consists of four three-slot windows, with consecutive windows overlapping by one slot. The first subtask of each task has a pseudo-deadline at slot 2. Because $b$ has been eliminated, this tie can be broken arbitrarily. We break it in favor of the subtasks of the tasks in set $A$. Therefore the eight tasks in set $A$ are scheduled in slots 0 and 1. In slot 2, the three tasks from set $B$ are the only tasks with subtasks that are eligible for execution. Hence, only three subtasks can be scheduled in slot 2, causing a deadline miss later at time 9. ■

The definition of $D(T_i)$ ensures that if $T$ is light and $U$ is heavy and if $d(T_i) = d(U_j) \wedge b(T_i) = b(U_j) = 1$, then $U_j$ has higher priority. The following theorem shows that it is necessary to tie-break such a situation in favor of the heavy task.

**Theorem 3.3.** *Suppose the definition of* $D$ *is changed as follows: if* $T$ *is light, then* $D(T_i)$ *is a randomly-selected value. Then, there exists a feasible task system that is not correctly scheduled.*

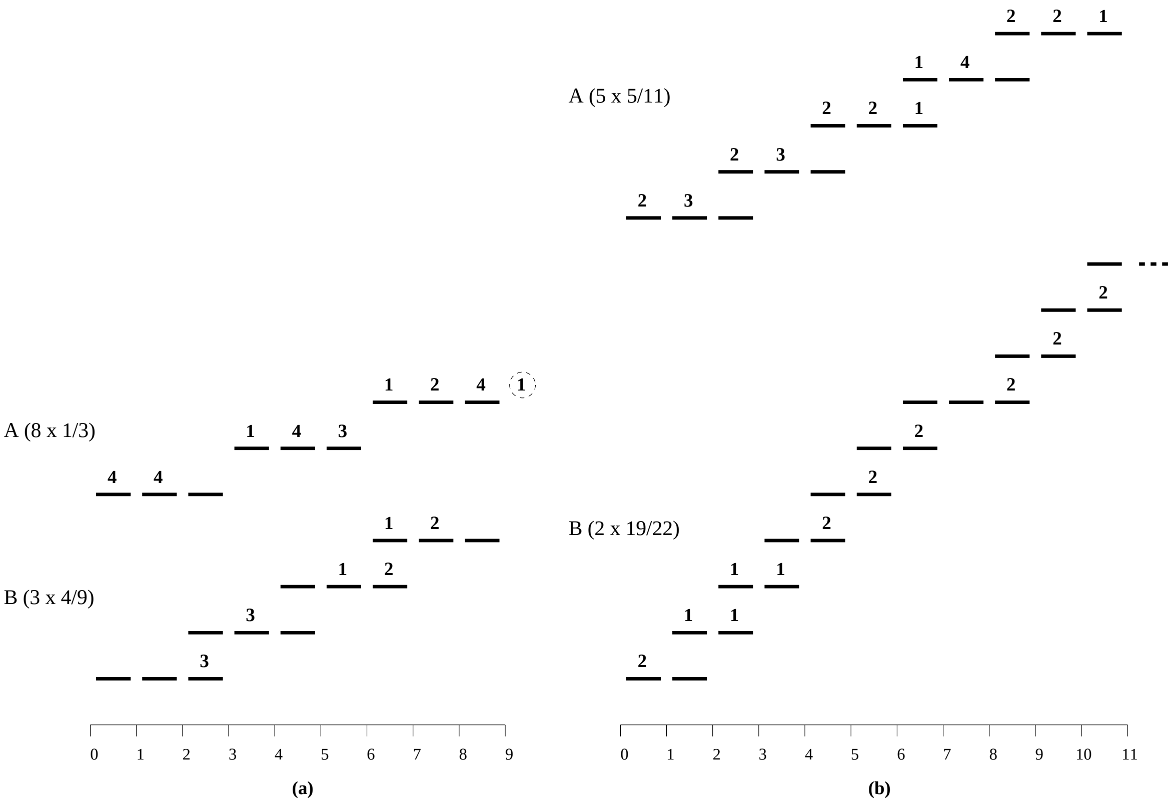



Figure 3.3: Tasks of a given weight are shown together. Each column corresponds to a time slot. For each subtask, there is an interval of time slots corresponding to its Pfair window (denoted by bold dashes). An integer value $n$ in slot $t$ means that $n$ of the tasks in the corresponding set have a subtask scheduled at $t$. No integer value means that no such subtask is scheduled in that slot. (We use the same notation in Figures 3.4, 3.5, 3.6, and 3.7.) **(a)** Theorem 3.2. A deadline is missed at time 9 by a task of weight 1/3. (We do not illustrate deadline misses in the subsequent figures, and show the schedule only until a slot is reached in which fewer than $M$ tasks are scheduled.) **(b)** Theorem 3.3.

*Proof.* Consider a task system consisting of a set $A$ of five light tasks with weight $5/11$ and a set $B$ of two heavy tasks with weight $19/22$. Because total utilization is four, this task system is feasible on four processors. Consider the schedule shown in Figure 3.3(b), which is possible given the proposed priority definition. In particular, at times 1, 3, and 7, the set-$A$ tasks are favored over the set-$B$ tasks. This causes only three subtasks to be eligible for execution in slot 10. ■

The previous counterexamples give rise to the possibility that $D(T_i)$ is actually only needed to tie-break heavy tasks over light tasks. The next theorem shows that this is

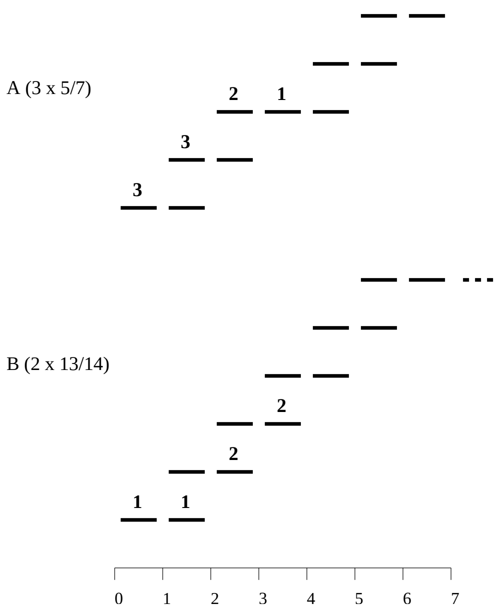



Figure 3.4: Theorem 3.4 (and Theorem 3.7).

not the case.

**Theorem 3.4.** *Suppose the definition of $D$ is changed as follows: if $T$ is heavy, then $D(T_i)$ is one. (If $T$ is light, then $D(T_i)$ is zero as before.) Then, there exists a feasible task system that is not correctly scheduled.*

*Proof.* Consider a task system, to be scheduled on four processors, consisting of a set $A$ of three heavy tasks with weight $5/7$ and a set $B$ of two heavy tasks with weight $13/14$. The proposed priority definition allows the schedule shown in Figure 3.4. Note that only three subtasks are eligible in slot 3. ■

Given the previous counterexample, one may wonder if the definition of $D$ can be weakened so that ties among heavy tasks are statically resolved. The following theorem shows that this is unlikely.

**Theorem 3.5.** *If $D$ is changed so that ties among heavy tasks are statically broken by weight, then there exists a feasible task system that is not correctly scheduled.*

*Proof.* Consider a task system, to be scheduled on 12 processors, consisting of a set $A$ of three heavy tasks with weight $8/9$ and a set $B$ of ten heavy tasks with weight $14/15$. First, suppose that $D$ is defined to statically tie-break the set-$A$ tasks over the set-$B$ tasks. Then, the schedule shown in Figure 3.5(a) is possible. In this schedule, only

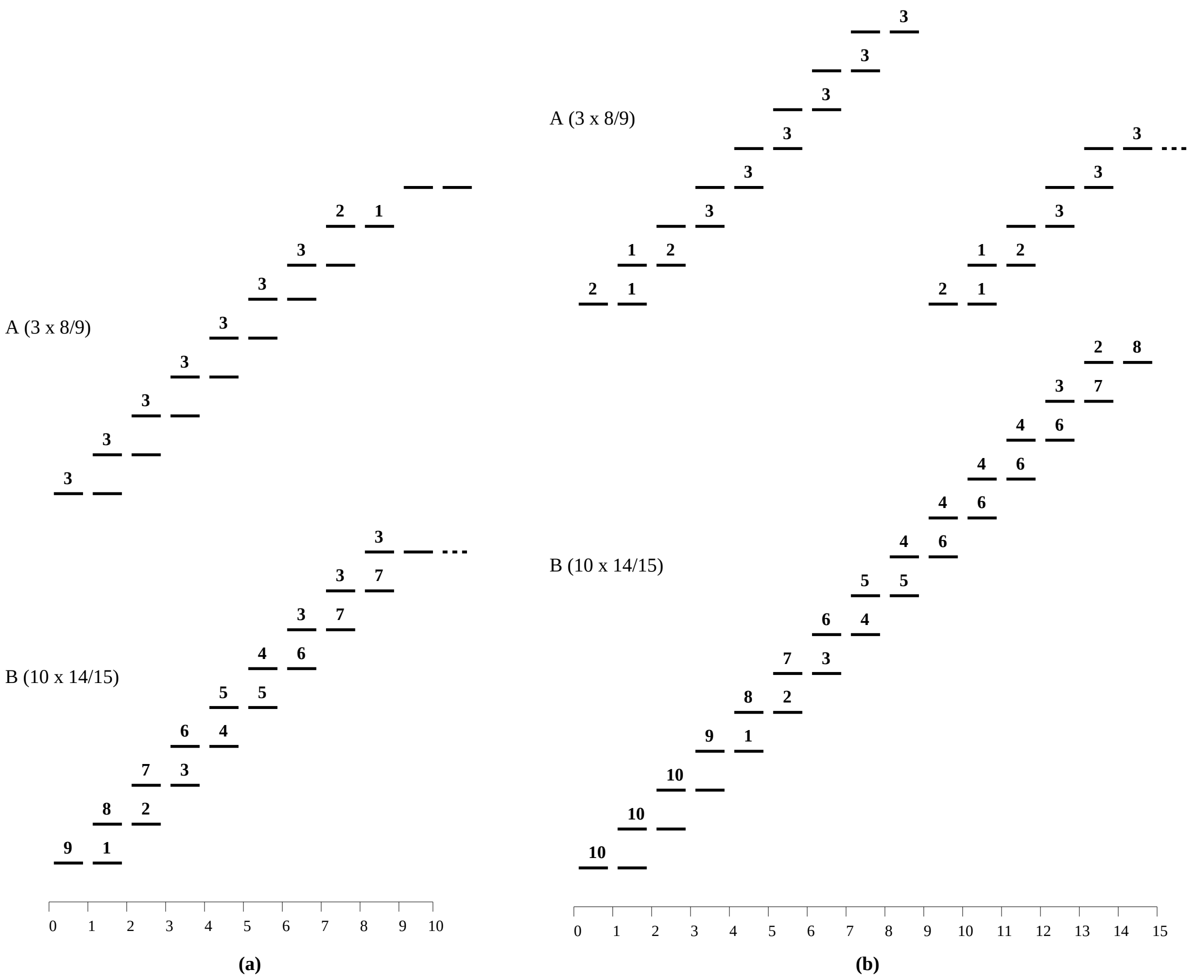



Figure 3.5: Theorem 3.5.

11 subtasks are eligible at time slot 8. Second, suppose that $D$ is defined to statically tie-break the set-$B$ tasks over the set-$A$ tasks. In this case, the schedule shown in Figure 3.5(b) is possible. In this schedule, only 11 subtasks are eligible at time slot 14. ■

From the previous theorem, it follows that $D$ *almost* certainly must be defined to dynamically tie-break heavy tasks. (Note, for example, that Theorem 3.5 leaves open the possibility of statically defining $D$ so that some set-$A$ tasks are favored over set-$B$ tasks, but other set-$A$ tasks are not favored over set-$B$ tasks.) One obvious approach to try that is less dynamic than ours is to define $D(T_i)$ based on the deadline of the current *job* of $T$. The next two theorems show that using job deadlines does not work; in the first of these theorems, later job deadlines are given higher priority, and in the

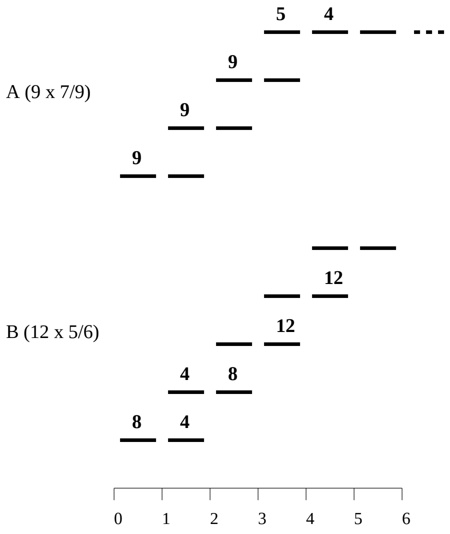



Figure 3.6: Theorem 3.6.

second, nearer job deadlines are given higher priority.

**Theorem 3.6.** *Suppose the definition of $D$ is changed as follows: if $T$ is heavy, then $D(T_i)$ is the deadline of the current job of $T$. Then, there exists a feasible task system that is not correctly scheduled.*

*Proof.* Consider a task system, to be scheduled on 17 processors, consisting of a set $A$ of nine heavy tasks with weight $7/9$ and a set $B$ of 12 heavy tasks with weight $5/6$. The proposed priority definition allows the schedule shown in Figure 3.6. (Note that the newly-proposed definition of $D$ favors set-$A$ tasks over set-$B$ tasks.) In this schedule, only 16 subtasks are eligible at time slot 4. ■

**Theorem 3.7.** *Suppose the definition of $D$ is changed as follows: if $T$ is heavy, then $D(T_i)$ is $1/t$, where $t$ is the deadline of the current job of $T$. Then, there exists a feasible task system that is not correctly scheduled.*

*Proof.* This can be proved by using the task system and schedule shown in Figure 3.4, which was used previously in the proof of Theorem 3.4. (Note that the newly-proposed definition of $D$ favors set-$A$ tasks.) ■

As we will show later in Chapter 5, neither $b$ nor $D$ is needed on two processors. We now show that at least one tie-breaking rule is needed in any system with three or more processors. (Note that Theorems 3.2, 3.3, 3.4 and 3.7 apply on systems with four or more processors.)

**Theorem 3.8.** *If our priority definition is changed by eliminating both $b$ and $D$, then there exists a task system that is feasible on three processors that is not correctly scheduled.*

*Proof.* Consider a task system, to be scheduled on three processors, consisting of a set $A$ of three heavy tasks with weight $1/2$ and a set $B$ of two heavy tasks with weight $3/4$. The proposed priority definition allows the schedule shown in Figure 3.7(a). In this schedule, only two subtasks are eligible at time slot 1. (Note that *either $b$ or $D$* would correctly tie-break these tasks.) ■

One "obvious" potential priority definition that comes to mind is to use the rational value $\frac{i}{wt(T)}$ as the deadline of subtask $T_i$, which is tantamount to omitting the ceiling brackets in the deadline formula (2.8). As it turns out, this new definition works for light-only task systems, but does not work if heavy tasks are present.

The optimality of this priority definition for light-only task systems follows from the optimality of PD$^2$. Recall that PD$^2$ only uses Rules (i) and (ii) for light tasks, *i.e.*, it prioritizes light tasks using the pair $(d(T_i), b(T_i))$. If $\frac{i}{wt(T)} \leq \frac{j}{wt(U)}$, then $\left\lceil \frac{i}{wt(T)} \right\rceil \leq \left\lceil \frac{j}{wt(U)} \right\rceil$, *i.e.*, $d(T_i) \leq d(U_j)$. Further, if $\frac{i}{wt(U)}$ is an integer, then $b(T_i) = 0$; in this case either $\frac{j}{wt(U)} > \frac{i}{wt(U)}$, in which case $d(U_j) > d(T_i)$ or $\frac{j}{wt(U)} = \frac{i}{wt(U)}$, in which case $b(U_j) = 0$. Thus, all the scheduling decisions are in accordance with PD$^2$.

It might appear that the expression $\frac{i}{wt(T)}$ reduces a task's priority to a single number. However, to avoid rounding errors, this rational number must be stored as two integers. PD$^2$ actually improves upon this by using an integer for $d(T_i)$ and a bit for $b(T_i)$.[1]

We now show that this new priority definition can sometimes fail to correctly schedule heavy tasks.

**Theorem 3.9.** *If the priority definition is changed so that $T_i$'s priority is at least $U_j$'s if $\frac{i}{wt(T)} \leq \frac{j}{wt(U)}$, and any ties are broken arbitrarily, then there exists a feasible task system such that it is not correctly scheduled.*

*Proof.* Consider a task system consisting of a set $A$ of 15 tasks with weight $3/5$ and a set $B$ of ten tasks with weight $9/10$. Total utilization is 18, so we should be able to schedule

[1] In fact, the PD$^2$ priority definition for light tasks can be reduced to a single integer $d'(T_i)$ that equals $2 \cdot d(T_i) - b(T_i)$. Under this new priority definition, $T_i$'s priority is at least $U_j$'s if $d'(T_i) \leq d'(U_j)$. It is straightforward to show that this algorithm makes the same scheduling decisions as PD$^2$ because the $b$-bit is either zero or one.

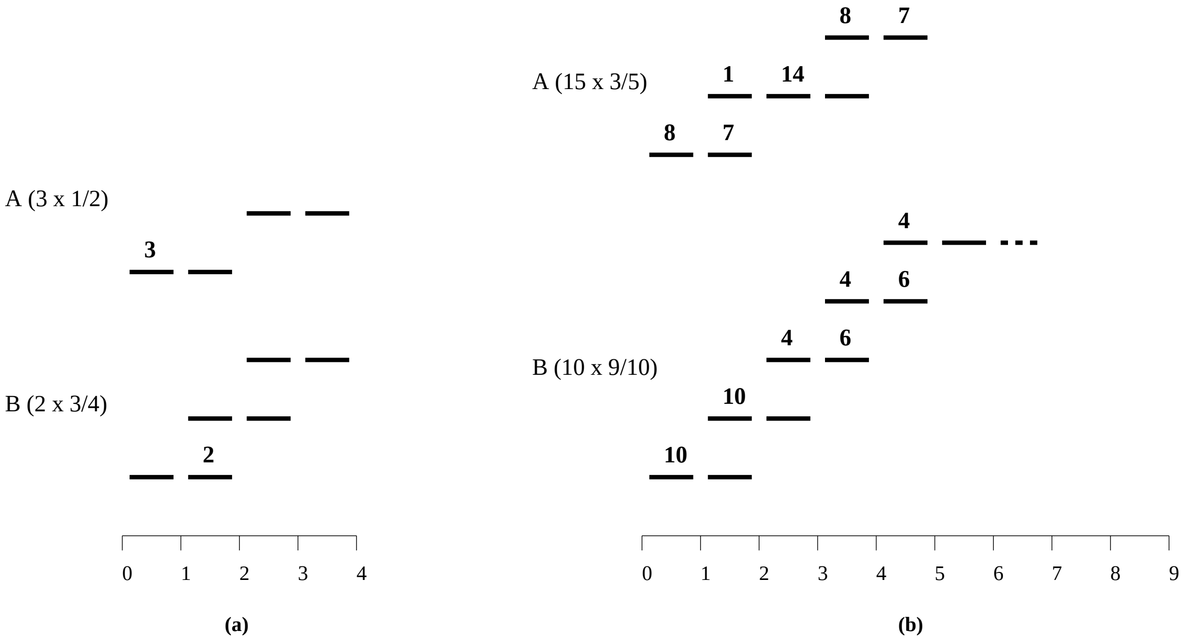



Figure 3.7: **(a)** Theorem 3.8. **(b)** Theorem 3.9.

this task system on 18 processors. Consider the schedule shown in Figure 3.7(b), which is possible given the proposed priority definition. In particular, at time 2, the priority of any task in $A$ is given by $\frac{2}{3/5} = \frac{10}{3}$, while the priority of a task in $B$ is given by $\frac{3}{9/10} = \frac{10}{3}$. Hence, tasks in set $A$ may be favored, as shown in Figure 3.7(b). This causes only 17 subtasks to be eligible for execution in slot 4. $\blacksquare$

## 3.3 Early-release Fair Scheduling

We now describe the concept of early-release fair (ERfair) scheduling; in Section 3.4, we prove that the ERfair version of $PD^2$ is optimal for scheduling periodic tasks on multiprocessors.

The notion of ERfair scheduling is obtained by simply dropping the $-1$ lag constraint from (2.6). Formally, a schedule is *early-release fair* (*ERfair*) if and only if

$$(\forall T, t :: lag(T, t) < 1). \tag{3.2}$$

In a *mixed* Pfair/ERfair-scheduled task system, each task's lag is subject to *either* (2.6) *or* (3.2); a task is called a *non-early-release* task in the former case, and an *early-release*

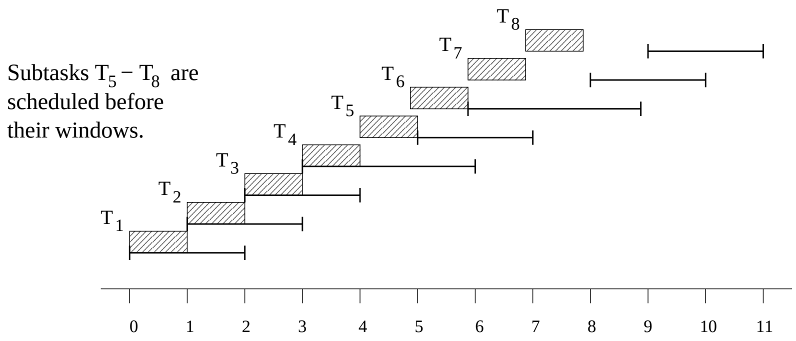



Figure 3.8: The Pfair windows of the first jobs of a task $T$ with weight 8/11 are shown. The schedule shown is ERfair, but not Pfair.

task in the latter. Note that any Pfair schedule is ERfair, but not necessarily vice versa.

Dropping the $-1$ lag constraint is equivalent to allowing subtasks to execute before their Pfair windows. Thus, in contrast to Pfair scheduling, a job that has not completed execution is always eligible for scheduling during its period under ERfair scheduling. More specifically, in a Pfair-scheduled system, a subtask $T_i$ is eligible at time $t$ if $t \in w(T_i)$, and if $T_{i-1}$ has been scheduled prior to $t$ but $T_i$ has not. In an ERfair-scheduled system, if $T_{i-1}$ and $T_i$ are part of the same job, then $T_i$ becomes eligible for execution immediately after $T_{i-1}$ executes, which may be before $T_i$'s Pfair window. This difference is illustrated in Figure 3.8. (Obviously, no subtask can become eligible before the beginning of the job that contains it. Hence, the first subtask of a job cannot be released early.) Note that ERfair scheduling is essentially a work-conserving extension to Pfair scheduling; as long as at least $M$ unfinished jobs exist, no processor is left idle under ERfair scheduling.

**Servicing aperiodic tasks.** As we shall see later in Chapter 6, one important application of mixed Pfair/ERfair scheduling is to permit the integrated scheduling of real-time periodic tasks and aperiodic tasks. (Recall that an aperiodic task is not recurrent and consists only of a single job.) The response times of aperiodic tasks can be improved by allowing server tasks to early-release their subtasks. This improves responsiveness without compromising the schedulability of the recurrent real-time tasks.

**Feasibility.** Because every Pfair schedule is also a valid ERfair schedule, (2.10) is a feasibility condition for ERfair-scheduled systems as well. For similar reasons, it is also a feasibility condition for mixed Pfair/ERfair-scheduled task systems.

**Scheduling algorithms.** PF, PD and $\text{PD}^2$ can be easily adapted to work for ERfair scheduling. The early-release versions are much simpler to implement, and are quite similar to more conventional priority-driven scheduling algorithms such as LLF. If $T_i$ is selected for execution, and if its successor $T_{i+1}$ is part of the same job, then $T_{i+1}$ is inserted immediately into the ready queue. If $T_i$ and $T_{i+1}$ are part of different jobs, then $T_{i+1}$ is inserted into an appropriate release queue, as described earlier for Pfair scheduling implementations. The early-release version is more efficient because fewer queue-merge operations need to be performed than in Pfair scheduling.

## 3.4 Optimality Proof of $\text{PD}^2$

We now prove that $\text{PD}^2$ correctly schedules any feasible *asynchronous* periodic task system. In an asynchronous periodic task system, each task may release its first job at any time. For Pfair- or ERfair-scheduled systems, this is equivalent to allowing the first subtask of each task $T$, namely $T_1$, to be released any time at or after time zero. Our notion of an asynchronous task system generalizes this: we allow a task $T$ to begin execution with any of its subtasks, perhaps one other than $T_1$, and this subtask may be released any time at or after time zero. As we shall see, this added generality facilitates our correctness proof for $\text{PD}^2$. It is straightforward to modify the flow construction used in the feasibility proof of (2.10) to apply to asynchronous task systems as defined here.[2] Thus, (2.10) is a feasibility condition for such systems. (In fact, the network flow construction produces a Pfair schedule, *i.e.*, each task is scheduled within its window.) In addition, it is possible to define the release and deadline of each subtask using simple formulae that are similar to those given in (2.7) and (2.8) for synchronous task systems. In particular, suppose task $T$ releases its first subtask at time $r$ and let $T_j$ $(j \geq 1)$ be this subtask. Then, the release and deadline of any subtask $T_i$ $(i \geq j)$ are given by the following formulae, where $\Delta(T) = r - \left\lfloor \frac{j-1}{wt(T)} \right\rfloor$.

$$r(T_i) = \Delta(T) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{3.3}$$

$$d(T_i) = \Delta(T) + \left\lceil \frac{i}{wt(T)} \right\rceil \tag{3.4}$$

[2] In fact, the feasibility proof for intra-sporadic task systems presented in Section 4.1 applies to asynchronous task systems as well.

Thus, the windows of $T$ are shifted by an *offset* given by $\Delta(T)$, which determines the release time of its first subtask. Note that if $T$ is an asynchronous periodic task according to the usual definition, then the first subtask released by $T$ is $T_1$ and $\Delta(T)$ reduces to $r$.

Before presenting the optimality proof of PD$^2$, we prove several properties that are used extensively in the proof. In the rest of this section, we assume that each task's weight is strictly less than one — a task with weight one would require a dedicated processor, and thus is quite easily scheduled. In other words, in this section, a heavy task $T$ has a weight in the range $[0.5, 1)$.

Note that by (3.3) and (3.4), the Pfair windows of two tasks $T$ and $U$ for which $\frac{T.e}{T.p} = \frac{U.e}{U.p}$ are identical. This implies that, under Pfair scheduling, they will be scheduled in precisely the same way. Thus, for notational simplicity, it is reasonable to assume that $T.e$ and $T.p$ are relatively prime for each task $T$, *i.e.*, $gcd(T.e, T.p) = 1$, where $gcd(a, b)$ is the greatest common divisor (GCD) of $a$ and $b$. *We do make this assumption in our proof.* Unfortunately, this creates a slight problem, because under ERfair scheduling, two tasks with equal weights but different periods may be scheduled differently. In particular, they may differ with regard to which subtasks may be released early because their job releases occur at different times. However, the above assumption is still valid because our proof applies even if subtasks are early-released across jobs. (In fact, our proof applies even if the scheduler *dynamically* decides whether to early release subtasks or not, or bounds early releases by a threshold — *e.g.*, a subtask may be allowed to release early, but only up to two time slots before its Pfair window.)

### 3.4.1 Properties About Subtask Windows

The following properties pertain to just a single task. For brevity, we let $T$ denote this task, and abbreviate $T.e$ and $T.p$ as $e$ and $p$, respectively. Thus, $wt(T) = \frac{e}{p}$, and (2.9) can be restated as follows. (Note that this formula remains the same even when equations (3.3) and (3.4) are used instead of (2.7) and (2.8), respectively.)

$$|w(T_i)| = \left\lceil \frac{ip}{e} \right\rceil - \left\lfloor \frac{(i-1)p}{e} \right\rfloor. \tag{3.5}$$

We now prove several properties about subtask windows and group deadlines.

**Lemma 3.1.** *The following properties hold for any task $T$.*

**(a)** *$r(T_{i+1})$ is either $d(T_i)$ or $d(T_i) - 1$, which implies that $r(T_{i+1}) \geq d(T_i) - 1$.*

**(b)** *The sequence of windows within any two jobs are identical, i.e.,* $|w(T_{ke+i})| = |w(T_i)|$*, where* $1 \le i \le e$ *and* $k \ge 0$.

**(c)** *The windows are symmetric within each job, i.e.,* $|w(T_{ke+i})| = |w(T_{ke+e+1-i})|$*, where* $1 \le i \le e$*, and* $k \ge 0$.

**(d)** *The length of each window is either* $\left\lceil \frac{p}{e} \right\rceil$ *or* $\left\lceil \frac{p}{e} \right\rceil + 1$.

**(e)** $|w(T_i)| = \left\lceil \frac{p}{e} \right\rceil$ *if* $(i-1)$ *is a multiple of* $e$.

*Proof.* Below, we prove each property separately.

**Proof of (a):** The required result follows because $r(T_{i+1}) = \Delta(T) + \left\lfloor \frac{i}{wt(T)} \right\rfloor$ (by (3.3)) and $d(T_i) = \Delta(T) + \left\lceil \frac{i}{wt(T)} \right\rceil$ (by (3.4)).

**Proof of (b):** By (3.5), $|w(T_{ke+i})| = \left\lceil \frac{(ke+i)p}{e} \right\rceil - \left\lfloor \frac{(ke+i-1)p}{e} \right\rfloor$. Therefore, $|w(T_{e+i})| = kp + \left\lceil \frac{ip}{e} \right\rceil - kp - \left\lfloor \frac{(i-1)p}{e} \right\rfloor = |w(T_i)|$.

**Proof of (c):** By part (b), we need to prove this only for the first job of $T$, *i.e.*, for $k = 0$. By (3.5),

$$
\begin{aligned}
|w(T_{e+1-i})| &= \left\lceil \frac{(e+1-i)p}{e} \right\rceil - \left\lfloor \frac{(e-i)p}{e} \right\rfloor \\
&= \left(p + \left\lceil \frac{(1-i)p}{e} \right\rceil\right) - \left(p + \left\lfloor \frac{-ip}{e} \right\rfloor\right) \\
&= \left\lceil \frac{(1-i)p}{e} \right\rceil - \left\lfloor \frac{-ip}{e} \right\rfloor \\
&= \left\lceil \frac{-(i-1)p}{e} \right\rceil + \left\lceil \frac{ip}{e} \right\rceil \\
&= -\left\lfloor \frac{(i-1)p}{e} \right\rfloor + \left\lceil \frac{ip}{e} \right\rceil
\end{aligned}
$$

Thus, $|w(T_i)| = |w(T_{e+1-i})|$.

**Proof of (d):** By (3.5), we have

$$
\begin{aligned}
|w(T_i)| &= \left\lceil \frac{ip}{e} \right\rceil - \left\lfloor \frac{(i-1)p}{e} \right\rfloor \\
&= \left\lceil \frac{ip}{e} \right\rceil - \left\lfloor \frac{ip}{e} - \frac{p}{e} \right\rfloor
\end{aligned}
$$

$$= \left\lceil \frac{ip}{e} \right\rceil + \left\lceil \frac{p}{e} - \frac{ip}{e} \right\rceil .$$

It is easy to see that this last expression equals either $\left\lceil \frac{p}{e} \right\rceil$ or $\left\lceil \frac{p}{e} \right\rceil + 1$.

**Proof of (e):** By (3.5), $|w(T_1)| = \left\lceil \frac{p}{e} \right\rceil$. By part (b), $|w(T_{ke+1})| = |w(T_1)|$. Thus, the required result follows.

■

The next property that we prove is used to prove property (P2) below, which is used several times in our proof. It refers to a "minimal" window of a task. Note that by part (d) of Lemma 3.1, the windows of any task are of at most two different lengths. We refer to a window of task $T$ with length $\left\lceil \frac{T.p}{T.e} \right\rceil$ as a *minimal* window of $T$. The following property follows by part (e) of Lemma 3.1.

**(P0)** The first window of each job of $T$ is a minimal window of $T$.

By (B), the $b$-bit of the last subtask of a job is zero. Therefore, the following property implies that the last window of each job of $T$ is a minimal window of $T$.

**(P1)** If $b(T_i) = 0$, then $|w(T_i)| = |w(T_{i+1})|$.

*Proof.* By (2.11), $b(T_i) = 0$ implies that $\frac{ip}{e}$ is an integer. Thus, because $gcd(e, p) = 1$, $i$ is a multiple of $e$. In other words, $i = (k+1)e$ for some $k \geq 0$, and $|w(T_i)| = |w(T_{ke+e})|$. Therefore, by part (c) of Lemma 3.1, we have $|w(T_i)| = |w(T_{ke+1})|$. By part (b) of Lemma 3.1, $|w(T_{ke+1})| = |w(T_{ke+e+1})|$. Therefore, $|w(T_i)| = |w(T_{i+1})|$, as required. ■

**(P2)** If $b(T_i) = 0$, then $w(T_i)$ is a minimal window of $T$.

*Proof.* As in the proof of (P1), we can show that $i$ is a multiple of $e$. Therefore, by (P0), $w(T_{i+1})$ is a minimal window. The required result then follows from (P1). ■

(P3) and (P4) below follow directly from part (d) of Lemma 3.1.

**(P3)** For all $i$ and $j$, $|w(T_j)| \leq |w(T_i)| + 1$.

**(P4)** For all $i$ and $j$, $|w(T_j)| \geq |w(T_i)| - 1$.

**(P5)** If $T$ is light, then all of its windows are of length at least three.

*Proof.* If $T$ is light, then $\frac{e}{p} < \frac{1}{2}$. Therefore, $\frac{p}{e} > 2$, and $\left\lceil \frac{p}{e} \right\rceil \geq 3$. The required result follows because $|w(T_i)| \geq \left\lceil \frac{p}{e} \right\rceil$ (by part (d) of Lemma 3.1). ■

**(P6)** $T$ has a 2-window if and only if it is heavy.

*Proof.* By part (e) of Lemma 3.1, $|w(T_1)| = \left\lceil \frac{p}{e} \right\rceil$. Note that $\left\lceil \frac{p}{e} \right\rceil$ is 2 if and only if $\frac{1}{2} \leq \frac{e}{p} < 1$, which implies that $T$ is heavy. ■

(P7) below follows directly from (P6) and part (d) of Lemma 3.1.

**(P7)** If $T$ is heavy, then all its windows are of length two or three.

The following property shows that the last subtask of each job of a heavy task has a 2-window.

**(P8)** If $T$ is heavy and $b(T_i) = 0$, then $|w(T_i)| = 2$.

*Proof.* By (P2), $|w(T_i)| = \left\lceil \frac{p}{e} \right\rceil$. Reasoning as in the proof of (P6), it follows that $|w(T_i)| = 2$. ■

**(P9)** If $t$ and $t'$ are successive group deadlines of a heavy task $T$, then $t' - t$ is either $\left\lceil \frac{1}{1 - wt(T)} \right\rceil$ or $\left\lceil \frac{1}{1 - wt(T)} \right\rceil - 1$.

*Proof.* By Theorem 3.1, the group deadlines of $T$ correspond to subtask deadlines of a task $U$ such that $wt(U) = 1 - wt(T)$. Therefore, by (3.4), $t' - t = \left\lceil \frac{j+1}{wt(U)} \right\rceil - \left\lceil \frac{j}{wt(U)} \right\rceil$ for some $j$. Thus, $t' - t = \left\lceil \frac{j}{wt(U)} + \frac{1}{wt(U)} \right\rceil - \left\lceil \frac{j}{wt(U)} \right\rceil$. From this, the required result follows. ■

The following claim is used to property (P10) below. (Refer to Figure 3.9.)

**Claim 3.1.** *Let $T$ be a heavy task with more than one group deadline per job. Let $t$ and $t'$ be consecutive group deadlines of $T$, where $t'$ is the first group deadline within some job of $T$ (for the first job of $T$, take $t$ to be 0). Similarly, let $u$ and $u'$ be consecutive group deadlines of $T$, where $u'$ is the last group deadline within some job of $T$. Then, $t' - t = u' - u + 1$.*

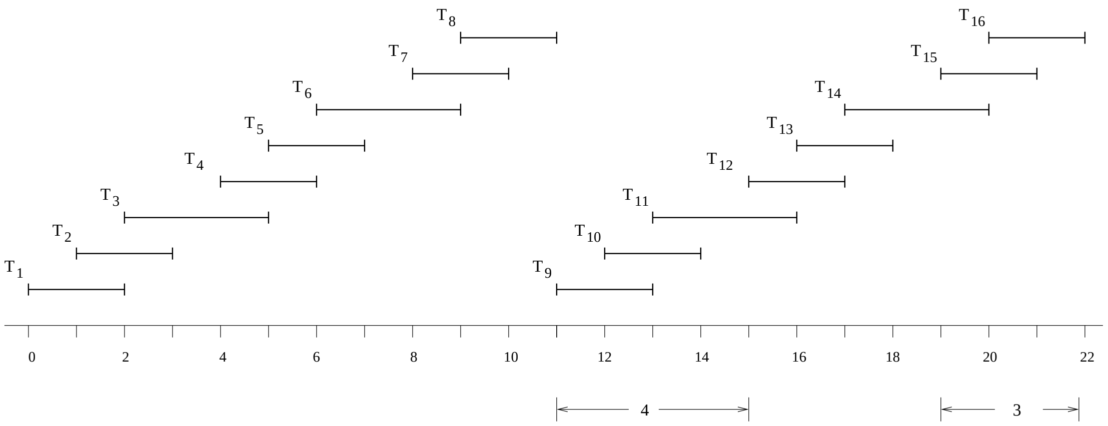



Figure 3.9: Windows of a task $T$ of weight $8/11$ are shown. Sample values of $t'$, $t$, $u$, and $u'$ (from Claim 3.1) are 11, 15, 19, 22.

*Proof.* By Theorem 3.1, the group deadlines of $T$ correspond to subtask deadlines of a task $U$ such that $U.e = p - e$ and $U.p = p$. Note that since $t'$ is the first group deadline within some job of $T$, $t$ is the last group deadline within the previous job, *i.e.*, $t$ corresponds to the deadline of a subtask $U_i$ such that $b(U_i) = 0$. By (2.11), $b(U_i) = 0$ implies that $\frac{ip}{p-e}$ is an integer. Note that because $gcd(e, p) = 1$, we have $gcd(p, p-e) = 1$. Therefore, $i$ is a multiple of $p - e$. In other words, $i = j(p - e)$ for some $j \geq 0$. (This also takes care of the case when $t = 0$.) Therefore, $d(U_i) = jp$, *i.e.*, $t = jp$.

Further, because $T$ has more than one group deadline per job, the number of subtask deadlines in each job of $U$ is at least 2, *i.e.*, $U.e \geq 2$. Therefore, $i + 1$ $(= j(p-e)+1)$ is not a multiple of $p - e$. Hence, $\left\lceil \frac{(j(p-e)+1)p}{p-e} \right\rceil = 1 + \left\lfloor \frac{(j(p-e)+1)p}{p-e} \right\rfloor$, which implies the following.

$$\left\lceil \frac{p}{p-e} \right\rceil = 1 + \left\lfloor \frac{p}{p-e} \right\rfloor \tag{3.6}$$

Also, we have $t' - t = \left\lceil \frac{(j(p-e)+1)p}{p-e} \right\rceil - jp = \left\lceil \frac{p}{p-e} \right\rceil$.

Similarly, we can show that $u' = kp$ for some $k$ and $u = \left\lceil \frac{(k(p-e)-1)p}{p-e} \right\rceil = kp + \left\lceil \frac{-p}{p-e} \right\rceil$. Therefore, $u' - u = -\left\lceil \frac{-p}{p-e} \right\rceil = \left\lfloor \frac{p}{p-e} \right\rfloor$.

Thus, $(t' - t) - (u' - u) = \left\lceil \frac{p}{p-e} \right\rceil - \left\lfloor \frac{p}{p-e} \right\rfloor$, which is 1 (by (3.6)). ∎

**(P10)** Let $T$ be a heavy task. Let $t$ and $t'$ be consecutive group deadlines of $T$, where $t$ is the last group deadline within some job of $T$ (for the first job of $T$, take $t$ to be 0).

Then $t' - t$ is at least the difference between any pair of consecutive group deadlines of $T$.

*Proof.* If $T$ has just one group deadline per job, then the difference between any two consecutive group deadlines exactly equals the period of $T$. On the other hand, if $T$ has multiple group deadlines within a job, then by Claim 3.1, $t' - t = u' - u + 1$, where $u'$ and $u$ are two consecutive deadlines of $T$. The required result then follows by (P9), because the difference between consecutive group deadlines can have at most two distinct values. ■

### 3.4.2 Optimality Proof

We now show that $\mathrm{PD}^2$ produces a "valid" schedule for any feasible asynchronous task system. A schedule is *valid at time slot t* if **(i)** for each subtask $T_i$ scheduled in slot $t$, $t$ lies within the interval during which $T_i$ is eligible (which implies that $T_i$ meets its deadline), **(ii)** no two subtasks of the same task are scheduled at $t$, and **(iii)** the number of tasks scheduled at $t$ is at most the number of processors. A schedule is *valid* if it is valid at every time slot. We begin by assuming, to the contrary, that $\mathrm{PD}^2$ fails to correctly schedule some task system. Then, there exists a time $t_d$ as follows.

**Definition 3.1.** *$t_d$ is the earliest time at which some feasible asynchronous task system misses a deadline under* $\mathrm{PD}^2$. ■

In other words, $\mathrm{PD}^2$ does not miss any deadline before time $t_d$ for any feasible asynchronous task system. Let $\tau$ be a feasible asynchronous task system with the following properties.

**(T1)** $\tau$ misses a deadline under $\mathrm{PD}^2$ at $t_d$.

**(T2)** Among all feasible task systems that miss a deadline under $\mathrm{PD}^2$ at $t_d$, no task system releases a larger number of subtasks in $[0, t_d)$ than $\tau$.

In the remainder of this section, we assume that $\tau$ is as defined here. The existence of such a $\tau$ follows from our assumption that $\mathrm{PD}^2$ is not optimal. We now show that $\mathrm{PD}^2$ produces a valid schedule for $\tau$ over $[0, t_d]$, thus contradicting our starting assumption.

In the proofs that follow, we consider slots in which one or more processors are idle. In a schedule $S$, if $k$ processors are idle at time slot $t$, then we say that there are $k$ *holes* in slot $t$ in $S$.

The following lemma gives an important property of the task set $\tau$. (Note that the proof of this lemma relies on (T2) and the fact that we have generalized the notion of an asynchronous system to allow a task to begin execution with any of its subtasks.)

**Lemma 3.2.** *If task $T \in \tau$ releases its first subtask at time $t > 0$, and if this first subtask is $T_i$, $i > 1$, then either $b(T_{i-1}) = 0$ and $|w(T_{i-1})| > t$ or $b(T_{i-1}) = 1$ and $|w(T_{i-1})| > t + 1$.*

*Proof.* We only consider the case when $b(T_{i-1}) = 0$ holds; in this case, we show that $|w(T_{i-1})| > t$ holds. (The proof for the case when $b(T_{i-1}) = 1$ holds is similar.) Suppose, to the contrary, that $|w(T_{i-1})| \leq t$. Consider the task system $\tau'$ obtained by adding the subtask $T_{i-1}$ with a release at time $t - |w(T_{i-1})| \geq 0$. (We assume that the relative priorities of two subtasks in $\tau$ do not change in $\tau'$.) Then, $\tau'$ satisfies the following properties.

- It has one more subtask than $\tau$.
- It misses a deadline at $t_d$.

To see the latter, note that upon adding $T_{i-1}$ to $\tau$, if $T_{i-1}$ does not miss its deadline, then it will either be scheduled in a slot where there is a hole, or it will cause a lower-priority subtask to be scheduled at a later slot. Inductively, this lower-priority subtask either misses a deadline or is scheduled correctly, in which case it may cause other subtasks to get scheduled later. Thus, no subtask will "shift" to an earlier slot. Repeating this argument, it is easy to see that adding $T_{i-1}$ cannot cause the missed deadline at $t_d$ to be met. Thus, $\tau'$ misses a deadline at $t_d$ or earlier. This contradicts either (T2) or the minimality of $t_d$ (refer to Definition 3.1). Therefore, $|w(T_{i-1})| > t$. ■

To avoid distracting boundary cases, we henceforth assume that the first subtask for each task $T$ is some $T_i$, where $i > 1$. This can be assumed without loss of generality, because if $T$ starts with $T_1$, then we can instead require it to start with $T_{x+1}$, where $x = T.e$; by part (b) of Lemma 3.1, the resulting release times and deadlines of the subtasks of $T$ will be identical, implying that the schedule produced by PD$^2$ is identical as well.

Our proof proceeds by showing the existence of certain schedules for task set $\tau$. To facilitate our description of these schedules, we find it convenient to totally order all subtasks in $\tau$. Let $\prec$ be an irreflexive total order that is consistent with the $\preceq$ relation in the PD$^2$ priority definition, *i.e.*, $\prec$ is obtained by arbitrarily breaking any ties left by $\preceq$.

**Definition 3.2.** *A schedule $S$ is defined to be $k$-compliant if and only if*

**(i)** *$S$ is valid,*

**(ii)** *the first $k$ subtasks according to $\prec$ are scheduled in accordance with* PD$^2$*, and*

**(iii)** *the remaining subtasks are scheduled within their Pfair windows* (*i.e., they are not early-released*). ■

We now present two lemmas. The second of these, Lemma 3.4, allows us to inductively prove that $\tau$ does not miss a deadline at $t_d$ as originally assumed. Lemma 3.3 deals with a situation arising in one of the cases in Lemma 3.4. According to Lemma 3.3, if subtasks $T_i$, $U_j$, and $U_{j+1}$ are scheduled as shown in Figure 3.10(a), then by swapping some subtasks, it is possible to obtain a schedule in which $U_j$ is not scheduled in slot $t$.

**Lemma 3.3.** *Let $S$ be a valid schedule for $\tau$ such that for light tasks $T$ and $U$ and $t < t'$, $U_j$ is scheduled in slot $t$, $T_i$ is eligible at $t$, and $U_{j+1}$ and $T_i$ are both scheduled in slot $t'$. Further, $r(U_j) = t$, $d(U_j) = t' + 1$, $r(U_{j+1}) = t'$, $d(T_i) = t' + 1$, and $w(U_j)$ is a minimal window of $U$. If all subtasks scheduled at or after $t$ in $S$ are scheduled within their Pfair windows, then there exists a valid schedule $S'$ also satisfying this property such that $U \notin S'_t$, $S_u = S'_u$ for $0 \leq u < t$, and $S_t - \{U\} \subset S'_t$.*

*Proof.* Our goal is to construct $S'$ by swapping $U_j$ with a later subtask. Unfortunately, $T_i$ and $U_j$ cannot be swapped directly because this would result in a schedule in which two subtasks of $U$ are scheduled in the same slot. Instead, we identify another subtask $V_k$ that can be used as an intermediate between $U_j$ and $T_i$ for swapping. Because $T$ and $U$ are both light, by (P5), all windows of each span at least three slots. Because $r(U_j) = t$ and $d(U_j) = t' + 1$, this implies that $t' \geq t + 2$ and $T$ and $U$ are not scheduled in slot $t' - 1$.

If there is a hole in slot $t' - 1$, then the swapping shown in Figure 3.10(b) gives the required schedule.

We henceforth assume that there is no hole in slot $t' - 1$. In this case, because $U$ is scheduled at $t'$ but not at $t' - 1$, there exists a task $V$ that is scheduled at $t' - 1$ but not at $t'$. Let $V_k$ be the subtask of $V$ scheduled at $t' - 1$. If $d(V_k) > t'$, then the swapping shown in Figure 3.10(c) gives the desired schedule. In the rest of this proof, we assume the following.

$$d(V_k) = t' \tag{3.7}$$

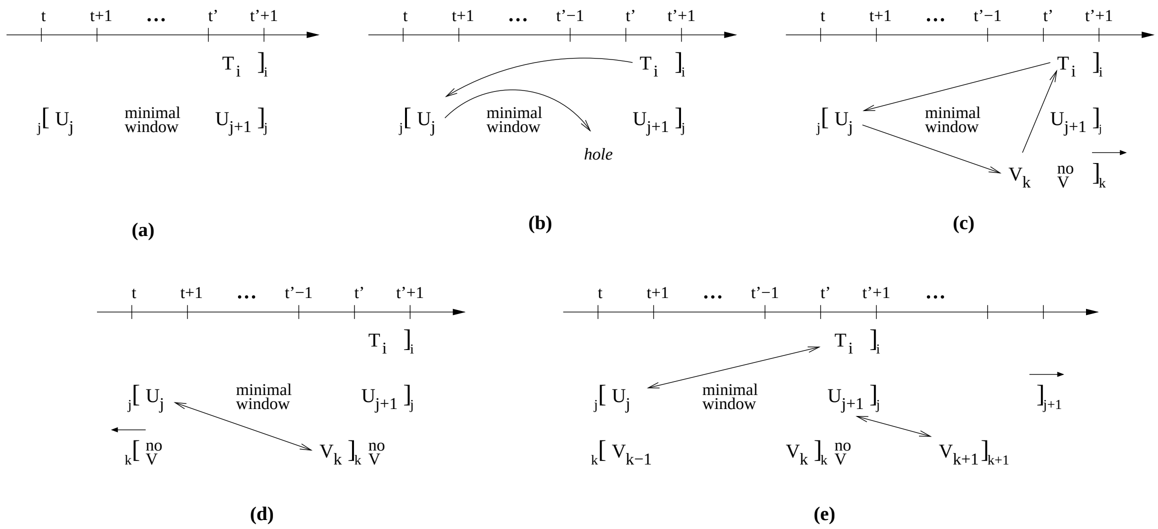



Figure 3.10: We use the following notation in this and the subsequent figures in this section. "[" and "]" indicate the release and deadline of a subtask; subscripts indicate which subtask. Each task is shown on a separate line. An arrow from subtask $T_i$ to subtask $U_j$ indicates that $T_i$ is now scheduled in place of $U_j$. An arrow over "[" (or "]") indicates that the actual position of "[" (or "]") can be anywhere in the direction of the arrow. Time is divided into unit-time slots that are numbered. (Although all slots are actually of the same length, due to formatting concerns, they do not necessarily appear as such in our figures.) If $T_i$ is released at slot $t$, then "[" is aligned with the left side of slot $t$. If $T_i$ has a deadline at time $t+1$, then "]" is aligned with the right side of slot $t$. In insets (c)–(e), no subtask of $V$ is scheduled in slot $t'$. **(a)** Conditions of Lemma 3.3. **(b)** There is a hole in slot $t'-1$. **(c)** $d(V_k) > t'$. **(d)** $d(V_k) = t'$ and $r(V_k) \leq t$. **(e)** $d(V_k) = t'$ and $r(V_k) \geq t$.

If $r(V_k) < t$ or if $r(V_k) = t \ \wedge\ V \notin S_t$, then the swapping shown in Figure 3.10(d) produces the desired schedule. The remaining possibility to consider is

$$(r(V_k) > t) \ \vee\ (r(V_k) = t \ \wedge\ V \in S_t). \tag{3.8}$$

In this case, we show that the swapping in Figure 3.10(e) is valid. (This inset actually depicts the case $r(V_k) = t \wedge V \in S_t$.) From (3.7) and the statement of the lemma, we have $d(V_k) = d(U_j) - 1$. Also, from (3.8), and the statement of the lemma, we have $r(V_k) \geq r(U_j)$. Therefore,

$$|w(V_k)| < |w(U_j)|. \tag{3.9}$$

Because $w(U_j)$ is a minimal window of $U$, $|w(U_{j+1})| \geq |w(U_j)|$. By definition, $|w(U_{j+1})| =$

$d(U_{j+1}) - r(U_{j+1})$, which implies that $d(U_{j+1}) = |w(U_{j+1})| + t'$. Therefore,

$$d(U_{j+1}) \geq t' + |w(U_j)|. \tag{3.10}$$

Now, by (3.7) and by part (a) of Lemma 3.1, $r(V_{k+1})$ is either $t'-1$ or $t'$. We now show that in either case, $d(V_{k+1}) \leq t' + |w(V_k)|$. If $r(V_{k+1}) = t'$ (in which case $b(V_k) = 0$), then by (P1), $|w(V_{k+1})| = |w(V_k)|$. By definition, $|w(V_{k+1})| = d(V_{k+1}) - r(V_{k+1})$. Therefore, $d(V_{k+1}) = t' + |w(V_k)|$.

On the other hand, if $r(V_{k+1}) = t' - 1$, then $|w(V_{k+1})| \leq |w(V_k)| + 1$ (by (P3)). Because $|w(V_{k+1})| = d(V_{k+1}) - r(V_{k+1})$, it follows that $d(V_{k+1}) \leq t' + |w(V_k)|$.

Thus, in both cases, we have $d(V_{k+1}) \leq t' + |w(V_k)|$. By (3.9) and (3.10), this implies that $d(U_{j+1}) > d(V_{k+1})$. Therefore, by part (a) of Lemma 3.1, $r(U_{j+2}) \geq d(V_{k+1})$. This implies that no subtask of $U$ is scheduled in the interval $[t'+1, d(V_{k+1}))$. Thus, the swapping shown in Figure 3.10(e) is valid, and produces the required schedule. ■

We now prove that a $k$-compliant schedule exists by induction on $k$. Note that a 0-compliant schedule is just a Pfair schedule (with no early releases), and the existence of such a schedule is guaranteed for any feasible task system. Also, if $n$ subtasks are released in $[0, t_d)$, then an $n$-compliant schedule is a valid schedule that is fully in accordance with PD$^2$ over $[0, t_d]$. The following lemma gives the inductive step of the proof.

**Lemma 3.4.** *If $S$ is a valid $k$-compliant schedule for $\tau$, then there exists a valid schedule $S'$ for $\tau$ that is $(k+1)$-compliant.*

*Proof.* Let $T_i$ be the $(k+1)^{st}$ subtask according to $\prec$. If $T_i$ is scheduled in $S$ in accordance with PD$^2$, then take $S'$ to be $S$. Otherwise, there exists a time slot $t$ such that $T_i$ is eligible at $t$ but scheduled later, and either **(i)** there is a hole in $t$, or **(ii)** some subtask ordered after $T_i$ by $\prec$ is scheduled at $t$. In the former case, we can easily rectify the situation by scheduling $T_i$ at $t$. Hence, in the rest of the proof, we assume that (ii) holds.

Let $t$ be the earliest such time slot, and let $U_j$ be the lowest-priority subtask scheduled at $t$. Thus, $T_i \prec U_j$. Let $t'$ be the slot where $T_i$ is scheduled, as depicted in Figure 3.11(a). Note that $t$ may or may not lie within $T_i$'s Pfair window; this depends on whether $T_i$ is an early-release subtask. However, because $S$ is $k$-compliant, $t$ lies within $U_j$'s Pfair window and $t'$ lies within $T_i$'s Pfair window. In the rest of the proof, we show that $S'$ can be obtained from $S$ by swapping $T_i$ and $U_j$ and perhaps some

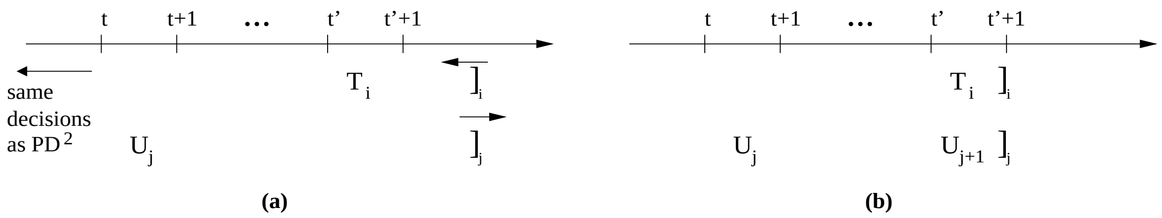



Figure 3.11: **(a)** Conditions of Lemma 3.4. **(b)** The "difficult" case to consider.

other subtasks. In all cases, the subtasks that are swapped include $T_i$ and subtasks ranked after $T_i$ by $\prec$. Since $S$ is $k$-compliant, all such subtasks are scheduled within their Pfair windows.

Because $S$ is a valid schedule and $T_i \prec U_j$, by the PD$^2$ priority definition, we have

$$t < t' < d(T_i) \leq d(U_j). \tag{3.11}$$

We first show that $U_{j+1}$ cannot be scheduled before slot $t'$. Because $T_i \prec U_j$, we have $T_i \prec U_{j+1}$, and hence $U_{j+1}$ is not early-released. Now, because $d(U_j) > t'$ (by (3.11)), by part (a) of Lemma 3.1, $r(U_{j+1}) \geq t'$. Thus, $U_{j+1}$ cannot be scheduled before $t'$. Further, it can be scheduled at $t'$ if and only if $r(U_{j+1}) = t'$.

If no subtask of $U$ is scheduled in the interval $[t+1, t'+1)$, then $T_i$ and $U_j$ can be directly swapped to get the required schedule. (Refer to Figure 2.7(a) in Section 2.2.2.) In the rest of the proof, we assume that $U_{j+1}$ is scheduled in slot $t'$ (*i.e.*, in the interval $[t', t'+1)$). As shown above, in this case $r(U_{j+1}) = t'$. Therefore, by part (a) of Lemma 3.1, $d(U_j)$ is either $t'$ or $t'+1$. By (3.11), it follows that $d(U_j) = t'+1$ and hence, $d(T_i) = t'+1$. Because $d(U_j) = r(U_{j+1}) + 1$, we have $b(U_j) = 1$. This implies that $b(T_i) = 1$, since $T_i \prec U_j$. Thus, we have the following.

$$U_{j+1} \in S_{t'} \ \wedge \ d(T_i) = d(U_j) = t'+1 \ \wedge \ r(T_{i+1}) = r(U_{j+1}) = t' \tag{3.12}$$

These conditions are depicted in Figure 3.11(b). We now consider four cases depending on the weights of $T$ and $U$.

**Case 1: $T$ is light and $U$ is heavy.** By the PD$^2$ priority definition and the definition of a group deadline, $T$ cannot have higher priority than $U$ at time $t$.

**Case 2: $T$ is heavy and $U$ is light.** In this case, we show that the swapping in Figure 3.12 is valid. (The argument hinges on the fact that $U$'s windows are at least as

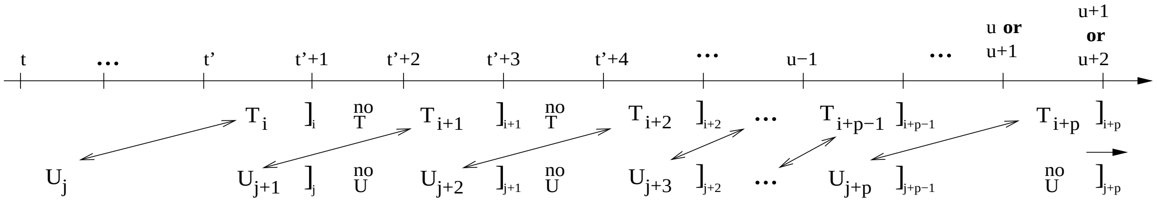



Figure 3.12: Case 2. $T$ is heavy, $U$ is light, and $d(T_i) = d(U_j)$.

long as $T$'s — see Figure 3.12.) By (P7), all windows of $T$ are of length either two or three. Further, by (P8), $|w(T_k)| = 2$ if $b(T_k) = 0$, and by (B) (refer to Section 2.2.2), $b(T_k) = 0$ if $T_k$ is the last subtask of a job. Because $b(T_i) = 1$, it follows that there exists an $r \geq 1$ such that

$$|w(T_{i+r})| = 2 \ \wedge \ (\forall k : 0 < k < r :: |w(T_{i+k})| = 3 \ \wedge \ b(T_{i+k}) = 1).$$

(Note that $r$ could be one, *i.e.*, $w(T_{i+1})$ could be a 2-window.) Because $U$ is light, by (P5), $|w(U_k)| \geq 3$ for all $k$. This implies that $d(T_{i+r}) < d(U_{j+r})$. Let $q$ denote the smallest value of $k$ that satisfies $d(T_{i+k}) < d(U_{j+k})$. (Note that $q \leq r$.) Then, $d(T_{i+q}) < d(U_{j+q})$, and for all $k \in [1, q-1]$,

$$d(T_{i+k}) = d(U_{j+k}) \ \wedge \ |w(T_{i+k})| = |w(U_{j+k})| = 3 \ \wedge \ b(T_{i+k}) = 1.$$

Because $d(T_{i+q}) < d(U_{j+q})$, by part (a) of Lemma 3.1, we have $d(T_{i+q}) \leq r(U_{j+q+1})$. Thus, $T_{i+q}$ is scheduled before $U_{j+q+1}$. Let $p$ be the smallest value for $k$ such that $T_{i+k}$ is scheduled prior to $U_{j+k+1}$. (Again, note that $p \leq q$). To summarize:

- $(\forall k \colon 0 < k < p :: d(T_{i+k}) = d(U_{j+k}) \wedge |w(T_{i+k})| = 3 \wedge |w(U_{j+k})| = 3 \wedge b(T_{i+k}) = 1) \ \wedge \ d(T_{i+p}) \leq d(U_{j+p})$,
- $T_{i+p}$ is scheduled before $U_{j+p+1}$, and
- for each $k$ in the range $0 < k < p$, $T_{i+k}$ is *not* scheduled before $U_{j+k+1}$.

It is straightforward to see that the relevant subtasks are scheduled as shown in Figure 3.12 and the depicted swapping is valid.

**Case 3: Both *T* and *U* are light.** (This case and Case 4 are somewhat lengthy.) Again, the situation under consideration is as depicted in Figure 3.11(b). Because $U$

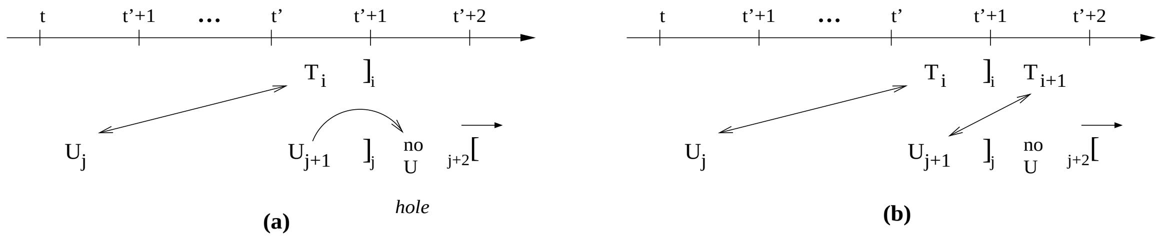



Figure 3.13: Case 3. **(a)** Some processor is idle in slot $t' + 1$. **(b)** $T_{i+1}$ is scheduled in slot $t' + 1$.

is light, by (P5), $|w(U_{j+1})| \geq 3$. Because $r(U_{j+1}) = t'$ (by (3.12)), this implies that $d(U_{j+1}) > t' + 2$. Therefore, by part (a) of Lemma 3.1, $r(U_{j+2}) \geq t' + 2$ and hence, $U$ is not scheduled in slot $t' + 1$. If there is a hole in slot $t' + 1$, then the swapping shown in Figure 3.13(a) gives the required schedule. Otherwise, if $T_{i+1}$ is scheduled in slot $t' + 1$, then the swapping shown in Figure 3.13(b) gives the required schedule. In the rest of Case 3, we assume that there is no hole in slot $t' + 1$ and $T_{i+1}$ is not scheduled there. We now show that one of the swappings shown in Figures 3.14 and 3.15 is valid.

Because $U$ is scheduled in slot $t'$ but not in slot $t' + 1$, and because there are no holes in slot $t' + 1$, there exists a task $V$ that is scheduled in slot $t' + 1$ but not in slot $t'$. Let $V_k$ be the subtask of $V$ scheduled in slot $t' + 1$. Because $S$ is a valid schedule, $d(V_k) \geq t' + 2$. Therefore, by (3.12), $d(V_k) > d(T_i)$, which implies that $T_i \prec V_k$. It follows that $V_k$ is not early-released and hence, $r(V_k) \leq t' + 1$. If $r(V_k) < t' + 1$, then the swapping shown in Figure 3.14(a) produces the desired schedule. In the rest of the proof for Case 3, we assume

$$r(V_k) = t' + 1, \tag{3.13}$$

in which case this swapping is not valid. By (3.12) and (3.13) we have the following.

$$r(V_k) = r(T_{i+1}) + 1 \tag{3.14}$$

We first dispense with the case $V_{k-1} \notin \tau$. In this case, by Lemma 3.2, either $(b(V_{k-1}) = 0 \ \wedge \ |w(V_{k-1})| > t' + 1)$ or $(b(V_{k-1}) = 1 \ \wedge \ |w(V_{k-1})| > t' + 2)$. In the former case, by (P1), $|w(V_k)| = |w(V_{k-1})|$; in the latter case, by (P4), $|w(V_k)| \geq |w(V_{k-1})| - 1$. Thus, in either case, $|w(V_k)| > t' + 1$. Further, since $T_i \in \tau$ and $d(T_i) = t' + 1$ (by 3.12), $|w(T_i)| \leq t' + 1$ (recall that slots are numbered from 0). Therefore, $|w(V_k)| > |w(T_i)|$, *i.e.*, $|w(V_k)| \geq |w(T_i)|+1$. By (P4), $|w(T_i)|+1 \geq |w(T_{i+1})|$. Thus, $|w(V_k)| \geq |w(T_{i+1})|$. Because $r(V_k) = r(T_{i+1}) + 1$ (by (3.14)), this implies that $r(V_k) + |w(V_k)| \geq r(T_{i+1}) + |w(T_{i+1})| + 1$. Therefore, $d(V_k) > d(T_{i+1})$, and hence no

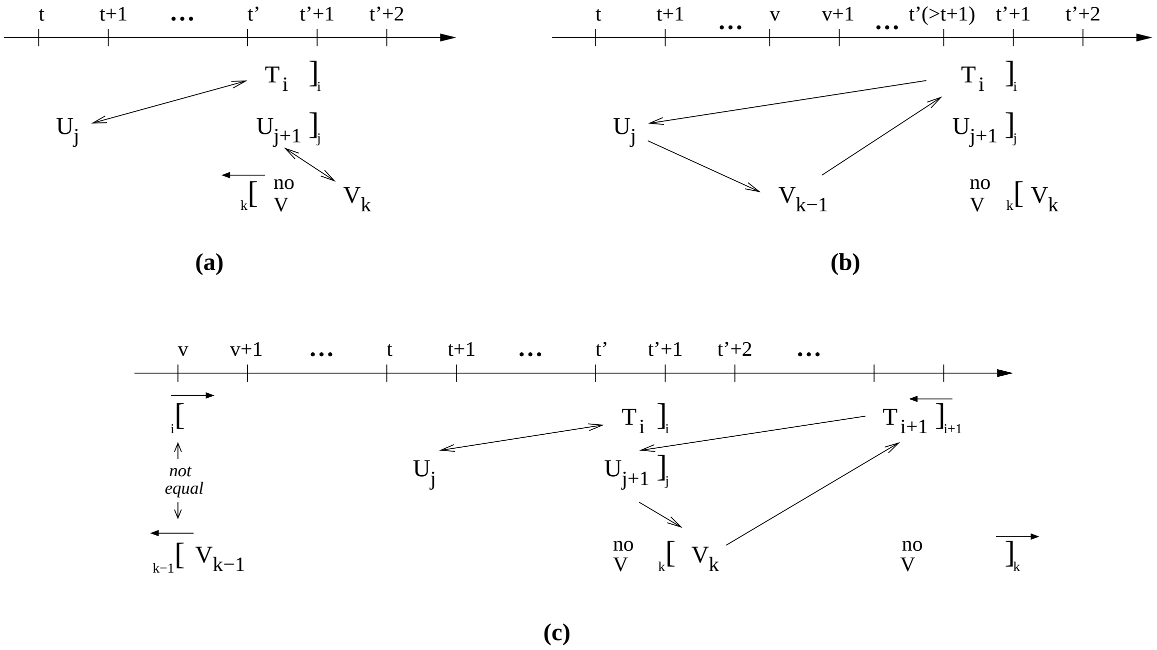



Figure 3.14: Case 3 (continued). **(a)** $r(V_k) \leq t'$. **(b)** $r(V_k) = t'+1$ and $V_{k-1}$ is scheduled at $v$, $t < v < t'$. **(c)** $r(V_k) = t'+1$ and $d(V_k) > d(T_{i+1})$.

subtask of $V$ is scheduled in $[t'+2, d(T_{i+1}))$. Thus, the swapping in Figure 3.14(c) is valid. (This figure actually depicts $V_{k-1} \in \tau$, but the swapping depicted is applicable nonetheless.) In the rest of the proof for Case 3, we assume that $V_{k-1} \in \tau$.

Note that because $r(V_k) = t'+1$ (by (3.13)), either $d(V_{k-1}) = t'+2$ or $d(V_{k-1}) = t'+1 \ \wedge \ b(V_{k-1}) = 0$. By (3.12), $d(T_i) = d(U_j) = t'+1$ and $b(T_i) = b(U_j) = 1$. Therefore, by the PD$^2$ priority definition, we have the following.

$$T_i \prec V_{k-1} \text{ and } U_j \prec V_{k-1} \tag{3.15}$$

If $V_{k-1}$ is scheduled in the interval $[t+1, t')$, then the swapping shown in Figure 3.14(b) is valid. If $V_{k-1}$ is not scheduled in $[t+1, t')$, then it is scheduled at or before $t$. Because $U_j \prec V_{k-1}$ (by (3.15)), $V_{k-1}$ is not scheduled in slot $t$, as this would contradict our choice of $U_j$ as the lowest-priority subtask scheduled at $t$.

In the rest of Case 3, we assume that $V_{k-1}$ is scheduled at a time $v < t$. Now, it must be the case that $T_i$ was not eligible to be scheduled at time $v$. To see this, note that if $T_i$ *were* eligible at time $v$, then it should have been scheduled there because $T_i \prec V_{k-1}$ (by (3.15)). This contradicts our starting assumption that $T_i$ should be scheduled at $t$. Thus, either $r(T_i) > v$ or $r(T_i) = v \ \wedge \ T_{i-1} \in S_v$. (Note that one of these assertions holds even if $T_i$ is an early-release subtask.) Because $T_i \prec V_{k-1}$ (by (3.15)), $V_{k-1}$ is not early-released. Therefore, $r(V_{k-1}) \leq v$, which implies that either $(r(T_i) > r(V_{k-1}))$ or $(r(T_i) = v \ \wedge \ r(V_{k-1}) = v \ \wedge \ T_{i-1} \in S_v)$. We consider these two subcases next.

**Subcase 3.A:** $r(T_i) > r(V_{k-1})$**.** We show that $d(V_k) > d(T_{i+1})$, which implies that the swapping shown in Figure 3.14(c) is valid. There are two possibilities to consider, depending on the value of $b(V_{k-1})$.

1. $b(V_{k-1}) = 0$. By (2.11), we have $d(V_{k-1}) = r(V_k) = t' + 1$ (by (3.13)). By (3.12), this implies that $d(V_{k-1}) = d(T_i)$. Since $b(V_{k-1}) = 0$, by (P1), $|w(V_k)| = |w(V_{k-1})|$. Because $r(V_{k-1}) < r(T_i)$ (our assumption for Subcase 3.A) and $d(V_{k-1}) = d(T_i)$, we have $|w(V_{k-1})| \geq |w(T_i)| + 1$. Therefore, $|w(V_k)| \geq |w(T_i)| + 1$. By (P4), $|w(T_i)| + 1 \geq |w(T_{i+1})|$, which implies that $|w(V_k)| \geq |w(T_{i+1})|$. Because $r(V_k) = r(T_{i+1}) + 1$ (by (3.14)), this implies that $r(V_k) + |w(V_k)| \geq r(T_{i+1}) + 1 + |w(T_{i+1})|$. Therefore, $d(V_k) > d(T_{i+1})$.

2. $b(V_{k-1}) = 1$. By the definition of $b(V_{k-1})$, we have $d(V_{k-1}) = r(V_k) + 1$. Hence, by (3.13), $d(V_k) = t' + 2$. By (3.12), this implies that $d(V_{k-1}) = d(T_i) + 1$. Along with $r(V_{k-1}) < r(T_i)$ (our assumption for Subcase 3.A), this implies that

$$|w(V_{k-1})| \geq |w(T_i)| + 2. \tag{3.16}$$

   By (P4), we have $|w(V_k)| \geq |w(V_{k-1})| - 1$ and $|w(T_i)| \geq |w(T_{i+1})| - 1$. Hence, by (3.16), $|w(V_k)| + 1 \geq |w(T_{i+1})| - 1 + 2$, *i.e.*, $|w(V_k)| \geq |w(T_{i+1})|$. Because $r(V_k) = r(T_{i+1}) + 1$ (by (3.14)), this implies that $d(V_k) > d(T_{i+1})$.

**Subcase 3.B:** $r(T_i) = v \ \wedge \ r(V_{k-1}) = v \ \wedge \ T_{i-1} \in S_v$**.** Reasoning as in Subcase 3.A, it follows that $d(V_k) \geq d(T_{i+1})$. By part (a) of Lemma 3.1, we have the following.

$$r(V_{k+1}) + 1 \geq d(V_k) \geq d(T_{i+1}) \tag{3.17}$$

We now show that a valid swapping exists in all cases. First, note that if $T_{i+1}$ is scheduled before $V_{k+1}$, then the swapping shown in Figure 3.14(c) is still valid. This will be the case if $r(V_{k+1}) \geq d(T_{i+1})$. In the rest of the proof for Subcase 3.B, we assume that $T_{i+1}$ is *not* scheduled before $V_{k+1}$. By (3.17), this can happen only if there exists a $v'$ that satisfies the following (see Figure 3.15).

$$r(V_{k+1}) = v' \ \wedge \ d(V_k) = v' + 1 \ \wedge \ d(T_{i+1}) = v' + 1 \ \wedge \ V_{k+1} \in S_{v'} \ \wedge \ T_{i+1} \in S_{v'} \quad (3.18)$$

The following property is used several times in the reasoning that follows.

**Claim 3.2.** *$w(V_k)$ is a minimal window of $V$.*

*Proof.* By part (a) of Lemma 3.1, (3.13) implies that $d(V_{k-1})$ is either $t'+1$ or $t'+2$. If $d(V_{k-1}) = t'+1$, then $b(V_{k-1}) = 0$ and by (P2), $w(V_k)$ is a minimal window. On the other hand, if $d(V_{k-1}) = t'+2$, then we have the following:

- $T_i$ and $V_{k-1}$ are both released at slot $v$ (our assumption for Subcase 3.B),
- $d(T_i) = t'+1$ (by (3.12)) and $d(V_{k-1}) = t'+2$,
- $r(T_{i+1}) = t'$ (by 3.12) and $r(V_k) = t'+1$ (by (3.13)), and
- $T_{i+1}$ and $V_k$ have equal deadlines (by (3.18)).

Therefore, $|w(T_i)| = |w(V_{k-1})| - 1$ and $|w(T_{i+1})| = |w(V_k)| + 1$. By (P3), $|w(T_{i+1})| \leq |w(T_i)| + 1$. Therefore, $|w(V_k)| + 1 \leq |w(V_{k-1})|$, *i.e.*, $|w(V_k)| \leq |w(V_{k-1})| - 1$. By (P4), this implies that $|w(V_k)| = |w(V_{k-1})| - 1$. By part (d) of Lemma 3.1, this implies that $w(V_k)$ is a minimal window of $V$. ■

To continue, if there is a hole in slot $v'-1$, then we can left-shift $T_{i+1}$ from $v'$ to $v'-1$ and apply the swapping shown in Figure 3.14(c). In the rest of Subcase 3.B, we assume that there is no hole in slot $v'-1$.

We now prove that $V$ must be a light task. Because $r(V_k) = t'+1$ (by (3.13)), $d(V_{k-1})$ is either $t'+1$ or $t'+2$, and if $d(V_{k-1}) = t'+1$, then $b(V_{k-1}) = 0$. Thus, because $r(T_i) = r(V_{k-1}) = v$ (our assumption for Subcase 3.B) and $d(T_i) = t'+1$ (by (3.12)), either $d(V_{k-1}) = d(T_i)+1$ or $d(V_{k-1}) = d(T_i) \ \wedge \ b(V_{k-1}) = 0$. Therefore, either $|w(V_{k-1})| = |w(T_i)|+1$ or $|w(V_{k-1})| = |w(T_i)| \ \wedge \ b(V_{k-1}) = 0$. Because $T$ is light, by (P5), $|w(T_i)| \geq 3$. Therefore, either $|w(V_{k-1})| \geq 4$ or $|w(V_{k-1})| \geq 3 \ \wedge \ b(V_{k-1}) = 0$. In either case, $V$ cannot be a heavy task: if it were heavy, then by (P7), all windows

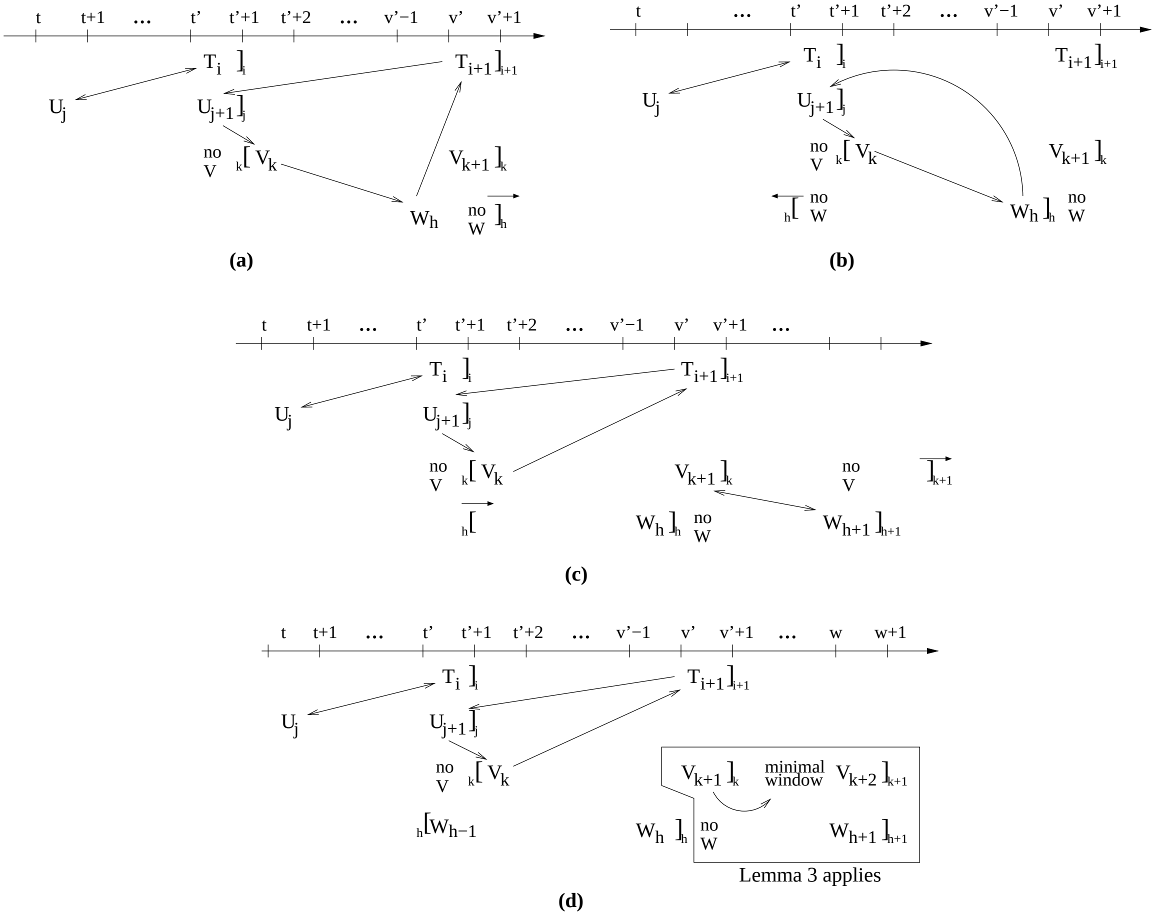



Figure 3.15: Subcase 3.B of Case 3. **(a)** $r(V_k) = t' + 1$, $d(V_k) = d(T_{i+1}) = v' + 1$, and $d(W_h) \geq v' + 1$. **(b)** $r(V_k) = t' + 1$, $d(V_k) = d(T_{i+1}) = v' + 1$, $d(W_h) = v'$, $r(W_h) \leq t'$, and $W$ is not scheduled at $t'$. **(c)** $r(V_k) = t' + 1$, $d(V_k) = d(T_{i+1}) = v' + 1$, $d(W_h) = v'$, and $r(W_h) > t'$ **(d)** $r(V_k) = t' + 1$, $d(V_k) = d(T_{i+1}) = v' + 1$, $d(W_h) = v'$, $r(W_h) = t'$, and $W$ is scheduled at $t'$.

of $V$ would be of length two or three, and by (P8), $b(V_{k-1}) = 0$ would imply that $|w(V_{k-1})| = 2$. Thus, $V$ is a light task.

By (P5), $|w(V_k)| \geq 3$. By (3.13) and (3.18), $w(V_k) = [t'+1, v'+1)$; hence, $v' \geq t'+3$. Because $V_k \in S_{t'+1}$ and $V_{k+1} \in S_{v'}$, this implies that $V \notin S_{v'-1}$. Thus, because there are no holes in slot $v' - 1$, there exists a task $W$ that is scheduled in slot $v' - 1$ but not in slot $v'$. Let $W_h$ be the subtask of $W$ scheduled in slot $v' - 1$. We now show that at least one of the swappings in Figure 3.15 is valid.

If $d(W_h) \geq v'+1$, then the swapping in Figure 3.15(a) is clearly valid. We henceforth assume

$$d(W_h) = v'. \tag{3.19}$$

If $(r(W_h) < t') \vee (r(W_h) = t' \wedge W \notin S_{t'})$, then the swapping shown in Figure 3.15(b) is valid. This leaves the following two possibilities.

1. $r(W_h) > t'$. In this case, we show that $d(W_{h+1}) < d(V_{k+1})$, which implies that the swapping in Figure 3.15(c) is valid. Because $r(W_h) > t'$, by (3.13), $r(W_h) \geq r(V_k)$. By (3.18) and (3.19), $d(W_h) < d(V_k)$. Therefore, $|w(W_h)| < |w(V_k)|$ (see Figure 3.15(c)). Because $w(V_k)$ is a minimal window of $V$ (by Claim 3.2), $|w(V_k)| \leq |w(V_{k+1})|$. Thus,

$$|w(W_h)| < |w(V_{k+1})|. \tag{3.20}$$

   Now, consider $b(W_h)$.

   If $b(W_h) = 0$, then by (3.19), $r(W_{h+1}) = v'$. Thus, by (3.18), $r(W_{h+1}) = r(V_{k+1})$. In addition, by (P1), $|w(W_{h+1})| = |w(W_h)|$. Hence, by (3.20), we have $|w(W_{h+1})| < |w(V_{k+1})|$. Therefore, $d(W_{h+1}) < d(V_{k+1})$.

   If $b(W_h) = 1$, then by (3.19), $r(W_{h+1}) = v' - 1$, which by (3.18) implies that $r(W_{h+1}) < r(V_{k+1})$. In addition, by (P3), $|w(W_{h+1})| \leq |w(W_h)| + 1$. Hence, by (3.20), we have $|w(W_{h+1})| \leq |w(V_{k+1})|$. Therefore, $d(W_{h+1}) < d(V_{k+1})$.

2. $r(W_h) = t' \wedge W \in S_{t'}$. In this case, analysis similar to that above shows that $d(W_{h+1}) \leq d(V_{k+1})$. Let $d(W_{h+1}) = w + 1$. If $d(W_{h+1}) < d(V_{k+1})$ or if $d(W_{h+1}) = d(V_{k+1}) \wedge V_{k+2} \notin S_w$, then the swapping shown in Figure 3.15(c) is valid. (The figure actually shows $W_h$ being released after time $t'$, but the swapping is still valid.) On the other hand, if $d(W_{h+1}) = d(V_{k+1})$ and $V_{k+2} \in S_w$, then we have the following (see Figure 3.15(d)).

   (a) $r(W_h) = t'$ and $r(V_k) = t' + 1$ (by (3.13)),

   (b) $d(W_h) = v'$ (by (3.19)) and $d(V_k) = v' + 1$ (by (3.18)),

   (c) $r(V_{k+1}) = v'$ (by (3.18)) and $r(W_{h+1})$ is either $v'$ or $v' - 1$ (by part (a) of Lemma 3.1 because $d(W_h) = v'$), and

   (d) $d(V_{k+1}) = d(W_{h+1}) = w + 1$.

   By (a) and (b) above, we have

$$|w(W_h)| = |w(V_k)|. \tag{3.21}$$

We now show that $|w(V_{k+1})| \leq |w(V_k)|$. If $r(W_{h+1}) = v' - 1$, then $|w(V_{k+1})| = |w(W_{h+1})| - 1$. By (P3), $|w(W_{h+1})| - 1 \leq |w(W_h)|$. Therefore, by (3.21), $|w(V_{k+1})| \leq |w(V_k)|$. On the other hand, if $r(W_{h+1}) = v'$, then $|w(V_{k+1})| = |w(W_{h+1})|$ and $b(W_h) = 0$ (because $d(W_h) = v'$, by (3.19)). Therefore, by (P1), $|w(W_{h+1})| = |w(W_h)|$. Thus, $|w(V_{k+1})| = |w(W_h)|$, and by (3.21), $|w(V_{k+1})| = |w(V_k)|$.

Because $w(V_k)$ is a minimal window of $V$ (by Claim 3.2), this implies that $w(V_{k+1})$ is a minimal window as well. Thus, by Lemma 3.3, there exists a schedule in which $V_{k+1}$ is not scheduled at time $v'$. The swapping shown in Figure 3.15(d) is therefore valid.

This completes Case 3.

**Case 4: Both $T$ and $U$ are heavy.** In the proof for this case, we refer to successive group deadlines of a task. The following notation will be used. If $g$ is a group deadline of task $X$, then $\mathit{pred}(X, g)$ (respectively, $\mathit{succ}(X, g)$) denotes the group deadline of task $X$ that occurs immediately before (respectively, after) $g$. For example, in Figure 2.6, $\mathit{pred}(T, 8) = 4$ and $\mathit{succ}(T, 8) = 11$.

As before, we are dealing with the situation depicted in Figure 3.11(b). Because $T_i$ has higher priority than $U_j$ at time $t$ according to PD$^2$, $D(U_j) \leq D(T_i)$. Recall that $T_i \in S_{t'}$ and $d(T_i) = t'+1$ (refer to (3.12) and Figure 3.11(b)), *i.e.*, $T_i$ is scheduled in the last slot of its window. By the definition of a group deadline, all subsequent subtasks with deadlines at or before $D(T_i)$ have windows of length two that overlap with the window of their predecessor subtask. Therefore, each such subtask is scheduled in the last slot of its window.

Let $u$ be the earliest time after $t'$ such that $U \notin S_u$. Because $U_{j+1}$ is scheduled in the first slot of its window, there exists a time before the group deadline of $U_j$ such that $U$ is not scheduled at that time. Thus, $u < D(U_j)$. Because $D(U_j) \leq D(T_i)$, this implies that $u < D(T_i)$. If $u < D(T_i) - 1$ or $u + 1 = D(T_i) \ \wedge \ T \in S_u$ holds then we have the following (refer to Figure 3.16).

- No subtask of $U$ is scheduled in slot $u$ (*i.e.*, in $[u, u + 1)$).
- In all slots in $[t', u + 1)$, a subtask of $T$ is scheduled in the last slot of its window.
- In all slots in $[t', u)$, a subtask of $U$ is scheduled in the first slot of its window.

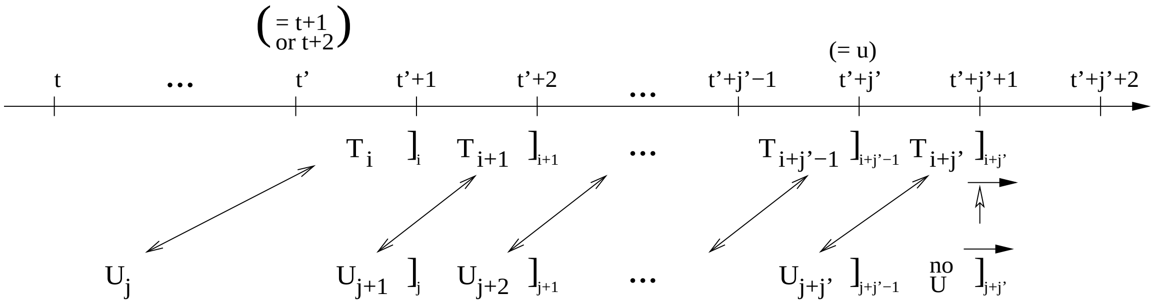



Figure 3.16: Case 4. We use the following notation in this figure and Figures 3.17–3.22. A group deadline at time $t$ is denoted by an up-arrow that is aligned with time $t$. A left- or right-pointing arrow over an up-arrow indicates a group deadline that may be anywhere in the direction of the arrow. $D(T_i) > D(U_j)$ or $u+1 = D(T_i) \ \wedge\ T \in S_u$.

This implies that the swapping in Figure 3.16 is valid.

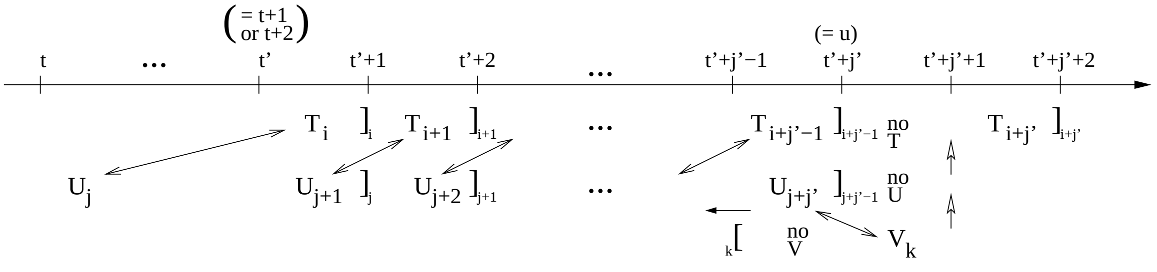
The remaining possibility is $u + 1 = D(T_i) \ \wedge\ T \notin S_u$. In this case, because $u+1 \leq D(U_j) \leq D(T_i)$, we have

$$D(T_i) = D(U_j) \ \wedge\ D(U_j) = u+1 \ \wedge\ T \notin S_u. \tag{3.22}$$

Let $U_{j+j'}$ be the subtask of $U$ scheduled at $u-1$. Then, $u = t' + j'$, as shown in Figure 3.17(a). By the definition of a group deadline, each of the subtasks $T_{i+1}, \ldots, T_{i+j'-1}$ and $U_{j+1}, \ldots, U_{j+j'-1}$ has a window of length two. (If not, $T_i$'s and $U_j$'s group deadlines will be earlier.) Therefore, $d(T_{i+j'-1}) = u$. By part (a) of Lemma 3.1, this implies that $r(T_{i+j'})$ is either $u$ or $u-1$. If $r(T_{i+j'}) = u$, then $b(T_{i+j'-1}) = 0$, which implies that $D(T_i) = u$. This contradicts (3.22). Therefore, $r(T_{i+j'}) = u-1$. Since $T_{i+j'-1}$ is scheduled in slot $u-1$ and $T \notin S_u$ (by (3.22)), it follows that $d(T_{i+j'})$ must be $u+2$. (By (P7), it cannot be later.) Thus, as shown in Figure 3.17(a), $w(T_{i+j'})$ is a 3-window starting at slot $u-1$, and $T_{i+j'}$ is scheduled in slot $u+1$, *i.e.*, slot $t'+j'+1$.

If there is a hole in slot $u$, then shifting subtask $U_{j+j'}$ to slot $u$ produces a situation in which a swapping similar to that in Figure 3.16 can be applied. We henceforth assume there is no hole in slot $u$.

Our strategy now is to identify another task to use as an intermediate for swapping. Because $T$ and $U$ are scheduled at $u-1$ but not at $u$, and because there are no holes in $u$, there exists a task $V$ that is scheduled at $u$ but not at $u-1$. Let $V_k$ be the subtask of $V$ scheduled at $u$. If $r(V_k) < u$, then the swapping in Figure 3.17(a) is valid, and if $r(V_k) = u \ \wedge\ V \notin S_{u+1}$, then the swapping in Figure 3.17(b) is valid. In the rest of the



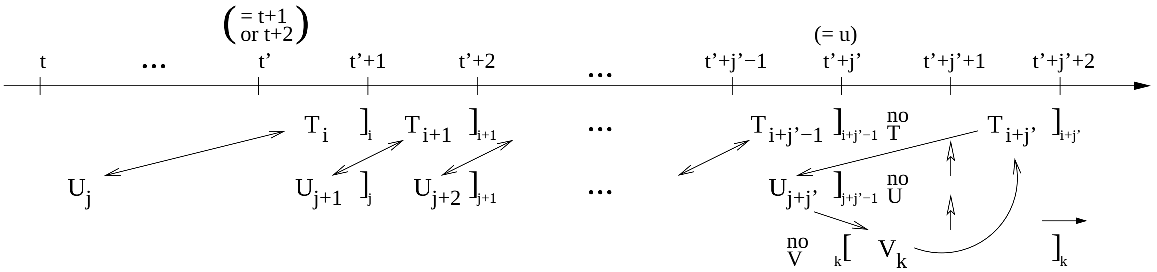




Figure 3.17: Case 4 (continued). **(a)** $D(T_i) = D(U_j)$ and $r(V_k) < t' + j'$. **(b)** $D(T_i) = D(U_j)$, $r(V_k) = t' + j'$, and $V \notin S'_{t'+j'+1}$.

proof, we assume

$$r(V_k) = u \ \wedge \ V \in S_{u+1}.$$

We now show that $V$ is a heavy task. Note that $V \in S_{u+1}$ implies that $r(V_{k+1}) \le u+1$. Therefore, by part (a) of Lemma 3.1, $d(V_k) \le u+2$. Because $r(V_k) = u$, by (P5) and (P7), $d(V_k) \ge u+2$. Therefore, we have $d(V_k) = u+2$, which implies that $|w(V_k)| = 2$. Therefore, by (P6), $V$ is heavy.

Consider $D(V_k)$, *i.e.*, the group deadline of $V_k$. Let $v$ be the earliest slot after $u$ such that $V \notin S_v$. Since $V_{k+1}$ is scheduled in the first slot of its window, we have

$$v + 1 \le D(V_k). \tag{3.23}$$

(See Figure 3.18.) Let $V_{k+i'}$ be the subtask of $V$ that is scheduled in slot $v-1$. If either $D(T_{i+j'}) > v+1$ or $D(T_{i+j'}) = v+1 \ \wedge \ b(T_{i+j'+i'}) = 0$, then $T_{i+j'+i'}$ is scheduled in slot $v$. To see why, note that $T_{i+j'}$ is scheduled in the last slot of its window and this forces all subtasks of $T$ until its group deadline to be scheduled in the last of their windows. Thus, the swapping shown in Figure 3.18 is valid. In the rest of the proof, we assume that neither of these conditions holds, *i.e.*, we assume the following.

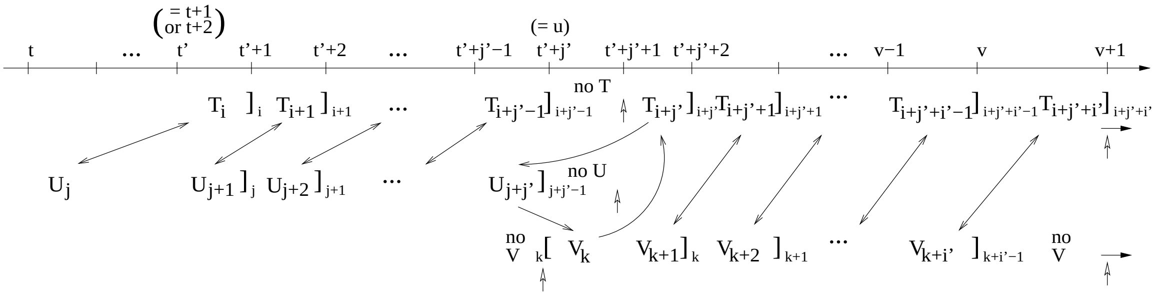


Figure 3.18: Case 4 (continued). $D(T_i) = D(U_j)$, $r(V_k) = t' + j'$, $D(T_{i+j'}) = v + 1$ and $T_{i+j'+i'}$ is scheduled in slot $v$.

$$(D(T_{i+j'}) < v + 1) \ \vee \ (D(T_{i+j'}) = v + 1 \ \wedge \ b(T_{i+j'+i'}) = 1) \tag{3.24}$$

We claim that $u$ is a group deadline of $V$. As seen in Figure 3.18, $V_{k-1}$ is not scheduled in slot $u - 1$. (Note that $V_{k-1} \in \tau$ because its window fits in the interval $[0, u + 1)$.) Because $r(V_k) = u$, by part (a) of Lemma 3.1, $d(V_{k-1})$ is either $u$ or $u+1$. If $d(V_{k-1}) = u$, then $b(V_{k-1}) = 0$ and hence, $u$ is a group deadline. If $d(V_{k-1}) = u + 1$, then we reason as follows. Because $V$ is a heavy task, by (P7), $|w(V_{k-1})| \leq 3$, which implies that $r(V_{k-1}) \geq u - 2$. Because $V_{k-1}$ is not scheduled in slots $u - 1$ or $u$, the following must hold.

$$r(V_{k-1}) = u - 2 \tag{3.25}$$

This implies that $w(V_{k-1}) = [u - 2, u + 1)$. Therefore, $u$ is a group deadline of $V$.

Having shown that $u$ is a group deadline of $V$, we now show that $pred(V, u) \leq pred(T, u+1)$. $T$ has consecutive group deadlines at $u+1$ and $succ(T, u+1) = D(T_{i+j'})$. Therefore, by (P9), the difference between $u+1$ and $pred(T, u+1)$ is at most one more than $D(T_{i+j'}) - (u + 1)$, *i.e.*, $u + 1 - pred(T, u + 1) \leq D(T_{i+j'}) - u$. Therefore,

$$pred(T, u + 1) \geq 2u - D(T_{i+j'}) + 1. \tag{3.26}$$

$V$ has consecutive group deadlines at $u$ and $succ(V, u) = D(V_k)$. Hence, by (P9), the difference between $u$ and $pred(V, u)$ is at least one less than $D(V_k) - (u)$, *i.e.*, $u - pred(V, u) \geq D(V_k) - u - 1$. Thus,

$$pred(V, u) \leq 2u - D(V_k) + 1. \tag{3.27}$$

By (3.23) and (3.24), $D(T_{i+j'}) \leq D(V_k)$. Therefore, by (3.26) and (3.27), $pred(V, u) \leq 2u - D(V_k) + 1 \leq 2u - D(T_{i+j'}) + 1 \leq pred(T, u+1)$. Thus, $pred(V, u) \leq pred(T, u+1)$.

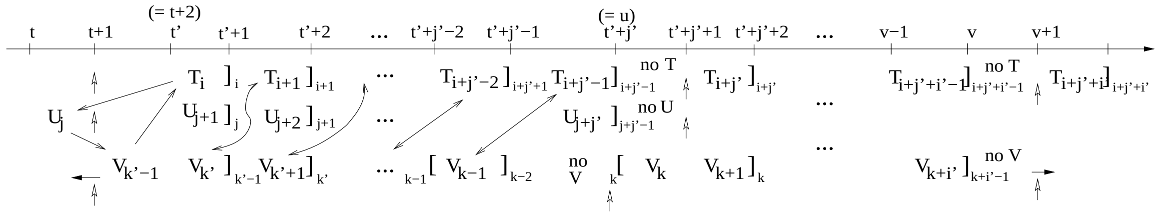


Figure 3.19: Subcase 4.A. $t' = t + 2$.

This sequence of inequalities further implies that $pred(V, u) = pred(T, u+1)$ if and only if $2u - D(V_k) + 1 = 2u - D(T_{i+j'}) + 1$, *i.e.*, $D(T_{i+j'}) = D(V_k)$. By (3.23) and (3.24), this can be true only if $D(T_{i+j'}) = D(V_k) = v + 1$.

In addition, as seen in Figure 3.18, $T$ cannot have a group deadline in the interval $(t', u]$. Therefore, we have the following.

$$pred(V, u) \leq pred(T, u + 1) \leq t' \tag{3.28}$$

$$(pred(V, u) = pred(T, u + 1)) \;\Rightarrow\; (D(T_{i+j'}) = v + 1 \;\wedge\; D(V_k) = v + 1) \tag{3.29}$$

Because $U$ is a heavy task, by (P7), $|w(U_j)| \leq 3$. Because $U_j$ is scheduled in slot $t$ and $T_i \prec U_j$, $U_j$ is not early-released, *i.e.*, $r(U_j) \leq t$. Further, because $d(U_j) = t' + 1$ (by (3.12)), it follows that $t + 1 < t' + 1 \leq t + 3$. Hence, $t'$ is either $t + 2$ or $t + 1$. We consider these two subcases next.

**Subcase 4.A: $t' = t + 2$.** In this case, we show that the swapping in Figure 3.19 is valid. To begin, note that $t' = t + 2$ implies that $T \notin S_{t+1}$ and $U \notin S_{t+1}$. Let $k' = k - j' + 1$. Then, we have the following as depicted in Figure 3.19.

- $r(V_{k-1}) = u - 2$ (by (3.25)) and $V_{k-1}$ is scheduled in slot $u - 2$.

- For each $l$ in the range $k' \leq l < k - 1$, $w(V_l)$ is a 2-window (since $pred(V, u) \leq t'$ by (3.28)).

- Each of $V_{k'}, \ldots, V_{k-1}$ is scheduled in the first slot of its window (which follows by inducting from right to left, starting with $V_{k-1}$). In particular, $V_{k'}$ is scheduled at $t' = r(V_{k'})$.

Before continuing, we note that all of the subtasks $V_{k'-1}, \ldots, V_{k-1}$ belong to $\tau$. To see why, note that their windows fit in the interval $[0, t' + j')$ (see Figure 3.19) and therefore, by Lemma 3.2, they are in $\tau$.

Because $V_{k'}$ is released at $t' = t+2$, by part (a) of Lemma 3.1, $d(V_{k'-1})$ is either $t+2$ or $t+3$. We now prove that $d(V_{k'-1}) \neq t+2$.

**Claim 3.3.** *$d(V_{k'-1}) \neq t+2$.*

*Proof.* Assume, to the contrary, that $d(V_{k'-1}) = t+2$. Because $r(V_{k'}) = t+2$ (refer to Figure 3.19 and the properties stated above), this implies that $b(V_{k'-1}) = 0$. Thus, by the definition of a group deadline,

$$pred(V, u) = t + 2. \tag{3.30}$$

Because $pred(V, u)$ corresponds to $d(V_{k'-1})$, and $b(V_{k'-1})$ is 0, it follows that $pred(V, u)$ is the last group deadline within some job of $V$. Therefore, by (P10), the difference between $u$ and $pred(V, u)$ is at least the difference between any pair of consecutive group deadlines of $V$. In particular, we have $succ(V, u) - u \leq u - pred(V, u)$. In either case, by (3.30), $succ(V, u) \leq 2u - t - 2$. Because $succ(V, u) = D(V_k)$, we have

$$D(V_k) \leq 2u - t - 2. \tag{3.31}$$

By (3.28), $pred(T, u+1) \leq t'$, *i.e.*, $pred(T, u+1) \leq t+2$. By (3.26), $D(T_{i+j'}) \geq 2u - pred(T, u+1) + 1$, which implies that $D(T_{i+j'}) \geq 2u - t - 1$. By (3.24), $v + 1 \geq D(T_{i+j'})$, and hence $v + 1 \geq 2u - t - 1$. By (3.23), we have $D(V_k) \geq v + 1$. Thus, $D(V_k) \geq 2u - t - 1$, which contradicts (3.31). Therefore, we conclude that $d(V_{k'-1})$ cannot be $t+2$. ■

Thus, we have the following.

$$d(V_{k'-1}) = t + 3$$

By (3.12), $d(T_i) = t' + 1 = t+3$. Further, $D(V_{k'-1}) = u < D(T_i)$ (by (3.22)). Therefore, $T_i \prec V_{k'-1}$.

We now conclude the proof for this subcase by showing that $V_{k'-1}$ is scheduled in slot $t+1$, which implies that the swapping in Figure 3.19 is valid. We have established that $V$ is heavy and $d(V_{k'-1}) = t+3$. Therefore, by (P7), all windows of $V$ are of length at most three, which implies that $r(V_{k'-1}) \geq t$. If $V_{k'-1}$ is not scheduled in slot $t+1$, then it must be scheduled at or before slot $t$. (Note that $V_{k'}$ is scheduled in slot $t+2$.)

Further, it is not early-released because $T_i \prec V_{k'-1}$ (proved in the preceding paragraph). Therefore, it must be scheduled in slot $t$ and $r(V_{k'-1}) = t$. (In other words, $w(V_{k'-1})$ must be a 3-window.) As seen in Figure 3.19, $d(V_{k'-1}) = d(U_j)$, $b(V_{k'-1}) = b(U_j) = 1$, and $D(V_{k'-1}) < D(T_i) = D(U_j)$. Thus, $U_j \prec V_{k'-1}$ contradicting our choice of $U_j$ as the lowest-priority subtask scheduled at $t$. Thus, $V_{k'-1}$ cannot be scheduled in slot $t$, and must be scheduled in slot $t+1$.

**Subcase 4.B: $t' = t + 1$.** In this case, we show that one of the swappings in Figures 3.20–3.22 is valid. We use the following result.

**Claim 3.4.** $pred(V, u) = t + 1$.

*Proof.* As in Subcase 4.A, we can show the following.

- $V_{k-1}$ has a window of length two or three and is scheduled in the first slot of its window.
- Each of $V_{k'}, \ldots, V_{k-2}$ has a window of length two and is scheduled in the first slot of its window. (The existence of subtasks $V_{k'}, \ldots, V_{k-1}$ in $\tau$ follows from the same reasoning given earlier in Subcase 4.A.)

This is depicted in Figure 3.20(a). The above facts, along with $t' = t + 1$, imply that $r(V_{k'}) = t + 1$. By part (a) of Lemma 3.1, this implies that $d(V_{k'-1})$ is either $t + 1$ or $t + 2$. If $d(V_{k'-1}) = t + 1$, then $b(V_{k'-1}) = 0$, and hence, $pred(V, u - 1) = t + 1$.

If $d(V_{k'-1}) = t + 2$, then we reason as follows. Because $V$ is heavy, by (P7), $w(V_{k'-1})$ is of length two or three. If $|w(V_{k'-1})| = 3$, then $w(V_{k'-1}) = [t - 1, t + 2)$ and hence, $pred(V, u) = t + 1$. Thus, it suffices to show that $|w(V_{k'-1})| \neq 2$.

Suppose, to the contrary, that $|w(V_{k'-1})| = 2$. (Note that, in this case, $V_{k'-1} \in \tau$ because its window fits in the interval $[0, t + 2)$.) Because $d(V_{k'-1}) = t + 2$, this implies that $r(V_{k'-1}) = t$. Note that $V_{k'-1}$ cannot have been early-released because $T_i \prec V_{k'-1}$ (this follows from Rule (iii) of the PD$^2$ priority definition because $D(V_{k'-1}) < D(T_i)$ — see Figure 3.20(a)). Because $V_{k'}$ is scheduled in slot $t+1$, $V_{k'-1}$ must be scheduled in slot $t$. Observe that $d(V_{k'-1}) = t + 2$, $d(U_j) = t' + 1 = t + 2$, $b(V_{k'-1}) = 1$, $b(U_j) = 1$,

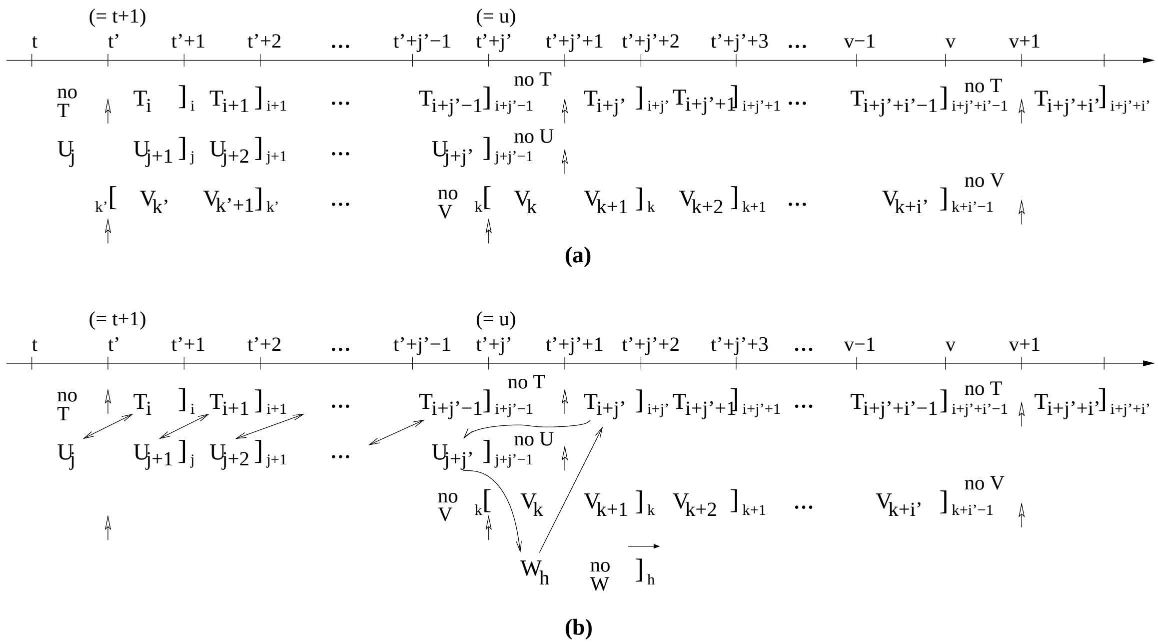



Figure 3.20: Subcase 4.B. In each inset of this figure and Figures 3.21 and 3.22, $t' = t+1$, $D(T_i) = D(U_j)$, and $D(V_k) = D(T_{i+j'})$. **(a)** $pred(V, u) = pred(T, u+1)$. **(b)** $d(W_h) \geq u+2$.

and $D(V_{k'-1}) < D(U_j)$ (again, refer to Figure 3.20(a)). Thus, $V_{k'-1}$ has lower priority than $U_j$ at $t$, which contradicts our choice of $U_j$ as the lowest-priority subtask scheduled at $t$. This completes the proof of Claim 3.4. ■

Because $t' = t + 1$, by (3.28) and Claim 3.4, we have

$$pred(T, u+1) = pred(V, u).$$

By (3.29), this implies that $D(T_{i+j'}) = D(V_k) = v + 1$ (see Figure 3.20(a)).

Because $T \notin S_u$ and $T \in S_{u+1}$, and because there are no holes in $u$, there exists a task $W$ that is scheduled at $u$ but not at $u+1$. Let $W_h$ be the subtask of $W$ scheduled at $u$. If $d(W_h) > u+1$, then the swapping shown in Figure 3.20(b) is valid. In the rest of the proof, we assume that

$$d(W_h) = u + 1.$$

In this case, we show that one of the swappings in Figures 3.21 and 3.22 are valid. If $W \notin S_{u-1}$, then the swapping shown in Figure 3.21(a) is valid. In the rest of the proof,

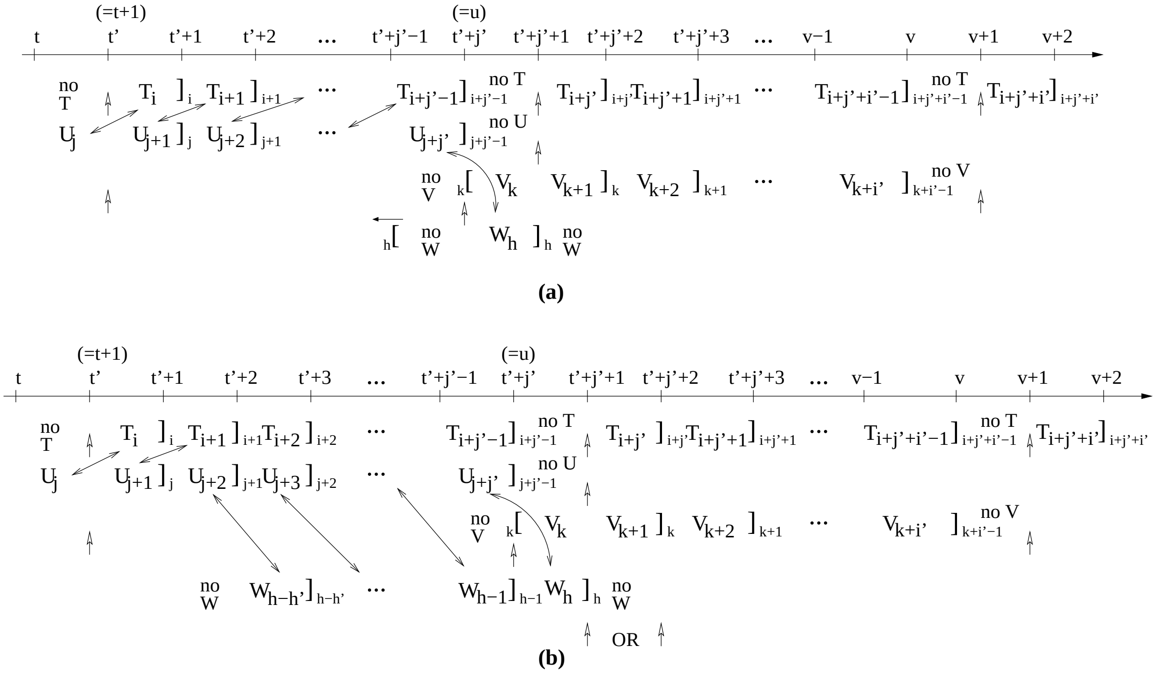



Figure 3.21: Subcase 4.B (continued). **(a)** $d(W_h) = u+1$ and $W \notin S_{u-1}$. **(b)** $d(W_h) = u+1$, $W \in S_{u-1}$, and $W \notin S_w$ for some $w$ in $[t', u)$.

we assume

$$W \in S_{u-1}.$$

In this case, we have $d(W_{h-1}) \geq u$. Because $d(W_h) = u+1$, this implies that $d(W_{h-1}) = u$ and $r(W_h) = u-1$. Thus, $|w(W_h)| = 2$. Thus, by (P6), $W$ is heavy.

We now show that $W$ has a group deadline at time $u+1$ or $u+2$ (refer to Figure 3.21(b)). Because $d(W_h) = u+1$, by by part (a) of Lemma 3.1, $r(W_{h+1})$ is either $u$ or $u+1$. If it is $u+1$, then $b(W_h) = 0$, *i.e.*, $W$ has a group deadline at $u+1$. If $r(W_{h+1})$ is $u$, then $d(W_{h+1})$ is either $u+2$ or $u+3$ (follows by (P7) because $W$ is heavy). Because no subtask of $W$ is scheduled in slot $u+1$, $W_{h+1}$ has to be scheduled in slot $u+2$ and $d(W_{h+1}) = u+3$. This implies that $w(W_{h+1})$ is a 3-window and hence $W$ has a group deadline at $u+2$.

We now look at earlier subtasks of $W$. If there exists a $w$ such that $t' \leq w \leq u-1$ and $W \notin S_w$, then a swapping similar to that shown in Figure 3.21(b) is valid and produces the desired schedule. In the rest of the proof, we assume that for each $w$ in the range $t' \leq w \leq u$, $W \in S_w$. This implies that, at each slot in the interval $[t', u+1)$, a subtask of $W$ is scheduled in the last slot of its window (recall that $W$ is heavy). This is illustrated in Figure 3.22. As seen in the figure, each of the subtasks $W_{h-j'+1}, \ldots, W_h$

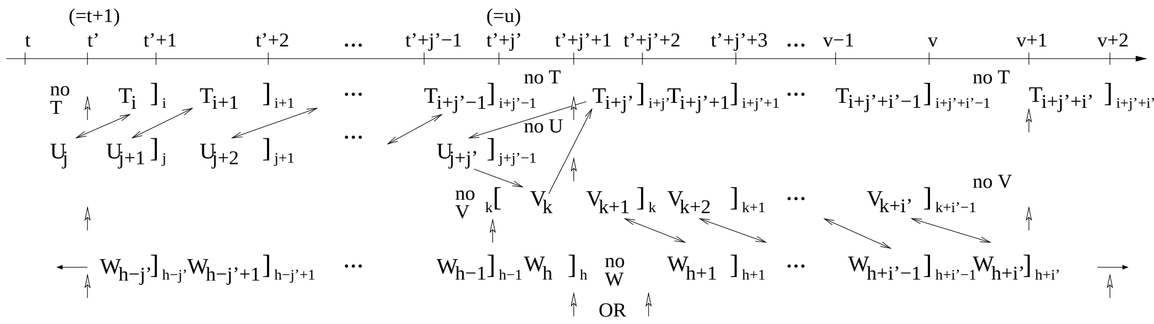


Figure 3.22: Subcase 4.B (continued). $d(W_h) = u+1$, $W \in S_{u-1}$, and $W$'s most recent group deadline before the one at $u+1$ or $u+2$ is at or before $t+1$.

must have a window of length two. This implies that the most recent group deadline of $W$ before the one at $u+1$ or $u+2$ occurs at or before time $t+1$, *i.e.*,

$$\begin{aligned}(u+1 \text{ is a group deadline of } W &\Rightarrow \mathit{pred}(W, u+1) \le t+1) \text{ and} \\ (u+2 \text{ is a group deadline of } W &\Rightarrow \mathit{pred}(W, u+2) \le t+1). \qquad (3.32)\end{aligned}$$

We now show that $W$'s next group deadline after the one at $u+1$ or $u+2$ occurs at or after time $v+2$, which implies that the swapping shown in Figure 3.22 is valid.

By (3.23) and (3.27), we have $\mathit{pred}(V, u) \le 2u - v$. Therefore, by Claim 3.4, we have the following.

$$v \le 2u - t - 1. \qquad (3.33)$$

There are now two possibilities to consider, depending on whether $W$ has a group deadline at $u+1$ or $u+2$.

1. $u+1$ is a group deadline of $W$. In this case, $b(W_h) = 0$. Thus, $u+1$ *is the last group deadline within some job of* $W$. Therefore, by (P10), the difference between $u+1$ and $\mathit{succ}(W, u+1)$ is at least the difference between any pair of consecutive group deadlines of $W$. In particular, $\mathit{succ}(W, u+1) - (u+1) \ge u+1 - \mathit{pred}(W, u+1)$. Thus, we have $\mathit{succ}(W, u+1) \ge 2u + 2 - \mathit{pred}(W, u+1)$.

   By (3.32), $\mathit{pred}(W, u+1) \le t+1$. Therefore, $\mathit{succ}(W, u+1) \ge 2u - t + 1$. By (3.33), this implies that $\mathit{succ}(W, u+1) \ge v+2$.

2. $u+2$ is a group deadline of $W$. In this case, by (P9), the difference between $\mathit{succ}(W, u+2)$ and $u+2$ is at least one less than the difference between $u+2$ and $\mathit{pred}(W, u+2)$, *i.e.*, $\mathit{succ}(W, u+2) - (u+2) \ge (u+2) - \mathit{pred}(W, u+2) - 1$. Therefore, $\mathit{succ}(W, u+2) \ge 2u - \mathit{pred}(W, u+2) + 3$.

By (3.32), $pred(W, u+2) \leq t+1$. Therefore, $succ(W, u+2) \geq 2u - t + 2$. By (3.33), this implies that $succ(W, u+2) \geq v+3$.

This exhausts all the possibilities if $T$ and $U$ are both heavy, and concludes the proof of Lemma 3.4. ■

By applying Lemma 3.4 inductively as discussed above, there exists a valid schedule for $\tau$ over $[0, t_d)$ consistent with PD$^2$, contrary to our original assumption. Thus, we have the following theorem.

**Theorem 3.10.** PD$^2$ *generates a valid schedule for any feasible asynchronous task system in which each task's lag is bounded by either* (2.6) *or* (3.2).

## 3.5 Experimental Comparison with Partitioning

In this section, we compare the PD$^2$ Pfair algorithm to the EDF-FF partitioning scheme, which uses "first fit" (FF) as a partitioning heuristic and EDF for per-processor scheduling. We begin by showing how to account for various system overheads in the schedulability tests for both approaches. We then present experimental results that show that PD$^2$ is a viable alternative to EDF-FF.

The feasibility tests described in Sections 2.1 and 2.2 were derived under the assumption that various overheads such as context-switching costs are zero. In practice, such overheads are usually not negligible; they can be accounted for by inflating task execution costs. In this section, we show how to do so for both PD$^2$ and EDF-FF. The specific overheads we consider are described next.

*Scheduling overhead* accounts for the time spent in moving a newly-arrived or preempted task to the ready queue and choosing the next task to be scheduled. Overheads associated with preemptions can be placed in two categories. *Context-switching overhead* accounts for the time the operating system spends on saving the context of a preempted task and loading the context of the task that preempts it. The *cache-related preemption delay* of a task refers to the time required to service cache misses that a task suffers when it resumes after a preemption. Note that scheduling and context-switching overheads are independent of the tasks involved in the preemption, whereas the cache-related preemption delay introduced by a preemption depends on **(i)** whether the preempted task resumes on a different processor, and **(ii)** which tasks execute in the meantime.

In a tightly-coupled, shared-memory multiprocessor, the cost of a migration is almost identical to that of a preemption. However, there might be some additional cache-related costs associated with a migration. If a task resumes execution on the same processor after a preemption (*i.e.*, without migrating), then some of the data that it accesses might still be in that processor's cache. This is highly unlikely if it resumes execution on a different processor. Nevertheless, because our analysis of cache-related preemption delays (described and justified later) assumes a cold cache after every preemption, there is no need to distinguish between preemptions and migrations.

All of the above overheads cause execution delays that must be considered when determining schedulability. In the rest of this section, we provide analytical worst-case bounds and empirical estimates of these various overheads for both EDF-FF and PD$^2$. We assume that PD$^2$ is invoked at the beginning of every quantum. When invoked, it executes on a single processor and then conveys its scheduling decisions to the other processors. Further, we assume that PD$^2$ schedules in a Pfair manner instead of ERfair; however, our results apply to the ERfair version of PD$^2$ as well.

**Context-switching overhead.** Under EDF, the number of preemptions is at most the number of jobs. Consequently, the total number of context switches is at most twice the number of jobs. Thus, context-switching overhead can be accounted for by simply inflating the execution cost of each task by $2c$, where $c$ is the cost of a single context switch. (This is a well-known accounting method.)

Under PD$^2$, a job may suffer a preemption at the end of every quantum in which it is scheduled. Hence, if a job spans $E$ quanta, then the number of preemptions suffered by it is bounded by $E - 1$. Thus, context-switching overhead can be accounted for by inflating the job's execution cost by $E \cdot c$. (The extra $c$ term bounds the context-switching cost incurred by the first subtask of the job.)

A better bound on the number of preemptions can be obtained by observing that when a task is scheduled in two consecutive quanta, it can be allowed to continue executing on the same processor. For example, consider a task $T$ with a period that spans 6 quanta and an execution time that spans 5 quanta. Then, in every period of $T$ there is only one quantum in which it is not scheduled. This implies that each job of $T$ can suffer at most one preemption. In general, a job of a task with period of $P$ quanta and an execution cost of $E$ quanta can suffer at most $P - E$ preemptions. Combining this with our earlier analysis, the number of context switches is at most $1 + \mathbf{min}(E - 1, P - E)$.

**Scheduling overhead.** Another concern regarding $\mathrm{PD}^2$ is the overhead incurred during each invocation of the scheduler. In every scheduling step, the $M$ highest-priority tasks are selected (if that many tasks have eligible subtasks), and the values of the release, deadline, $b$-bit, and group deadline for each scheduled task are updated. Also, an event timer is set for the release of the task's next subtask. When a subtask is released, it is inserted into the ready queue. In the case of EDF-FF, the scheduler on each processor selects the highest-priority job from its local queue; if this job is not the currently-executing job, then the executing job is preempted. If the executing job has completed, then it is removed from the ready queue. Further, when a new job is released, it is inserted into the ready queue and the scheduler is invoked.

The partitioning approach has a significant advantage on multiprocessors, since scheduling overhead does not increase with the number of processors. This is because each processor has its own scheduler, and hence the scheduling decisions on the various processors are made independently and in parallel. On the other hand, under $\mathrm{PD}^2$, the decisions are made sequentially by a single scheduler. Hence, as the number of processors increase, scheduling overhead also increases.

To measure the scheduling overhead incurred, we conducted a series of experiments involving randomly-generated task sets. Figure 3.23 compares the average[3] execution time of one invocation of $\mathrm{PD}^2$ with that of EDF on a *single* processor. We used binary heaps to implement the priority queues of both schedulers. The number of tasks, $N$, ranged over the set $\{15, 30, 50, 75, 100, 250, 500, 750, 1000\}$; for each $N$, 1,000 task sets were generated randomly, each with total utilization at most one. Then, each generated task set was scheduled using both $\mathrm{PD}^2$ and EDF until time $10^6$ to determine the average execution cost per invocation for each scheduler.

As the graph shows, the scheduling overhead of each algorithm increases as the number of tasks increases. Though the increase for $\mathrm{PD}^2$ is higher, the overhead is still less than 8 $\mu s$. When the number of tasks is at most 100, the overhead of $\mathrm{PD}^2$ is less than 3 $\mu s$, which is comparable to that of EDF.

Figure 3.24 shows the average scheduling overhead of $\mathrm{PD}^2$ for $2, 4, 8$, and 16 processors. Again, the overhead increases as more tasks or processors are added. However, the scheduling cost for at most 200 tasks is still less than 20 $\mu s$, even for 16 processors. The cost of a context switch in modern processors is between 1 and 10 $\mu s$ [Raj02].

[3]The experiments were performed on a 933-MHz Linux platform in which the minimum accuracy of the clock is $10ms$. Since the various scheduling overheads are much less than this value, averages are presented instead of maximum values.

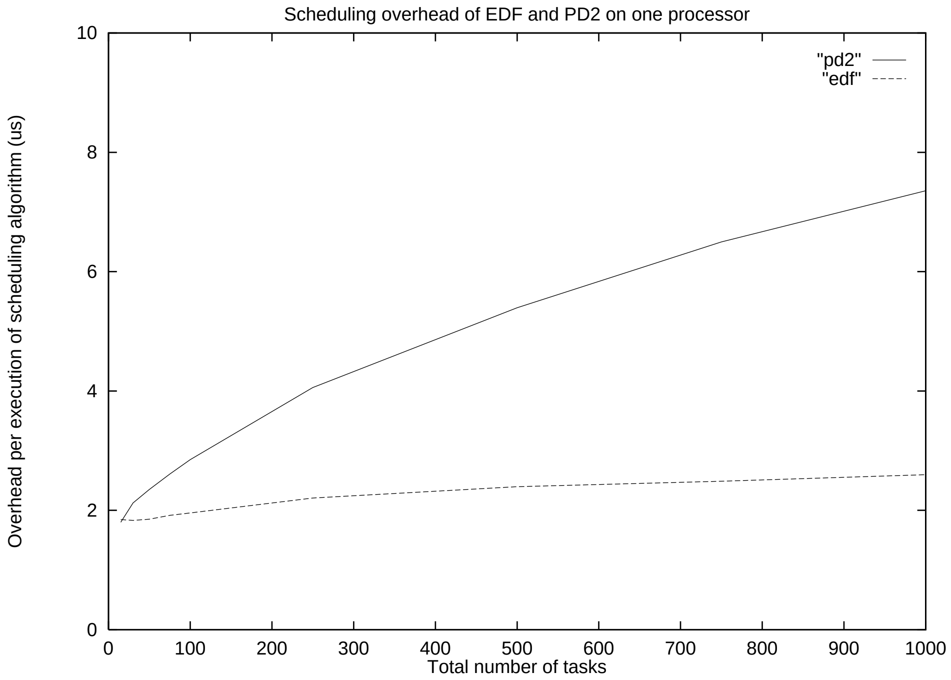



Figure 3.23: Scheduling overhead of EDF and PD$^2$ on one processor. 99% confidence intervals were computed for each graph but are not shown (in this and Figure 3.24) because the relative error associated with each point is very small — less than 1.2% of the reported value.

Thus, in most cases, scheduling costs are comparable to context-switching overheads under PD$^2$.

Scheduling overhead can be incorporated into a task's execution cost in much the same way as context-switching overhead. However, under PD$^2$, a job with an execution cost of $E$ quanta incurs a scheduling cost at the beginning of every quantum in which it is scheduled. Hence, this cost is incurred exactly $E$ times.

**Cache-related preemption delay.** For simplicity, we assume that *all* cached data accessed by a task is displaced upon a preemption or migration. Though this is clearly very conservative, it is difficult to obtain a more accurate estimate of cache-related preemption delay. This problem has been well-studied in work on timing analysis tools for RM-scheduled uniprocessors and several approaches have been proposed for obtaining better estimates [LHS$^+$98, LLH$^+$01]. Unfortunately, no such techniques are available for either EDF or PD$^2$.

The cache-related preemption delay under EDF can be calculated as follows. Let

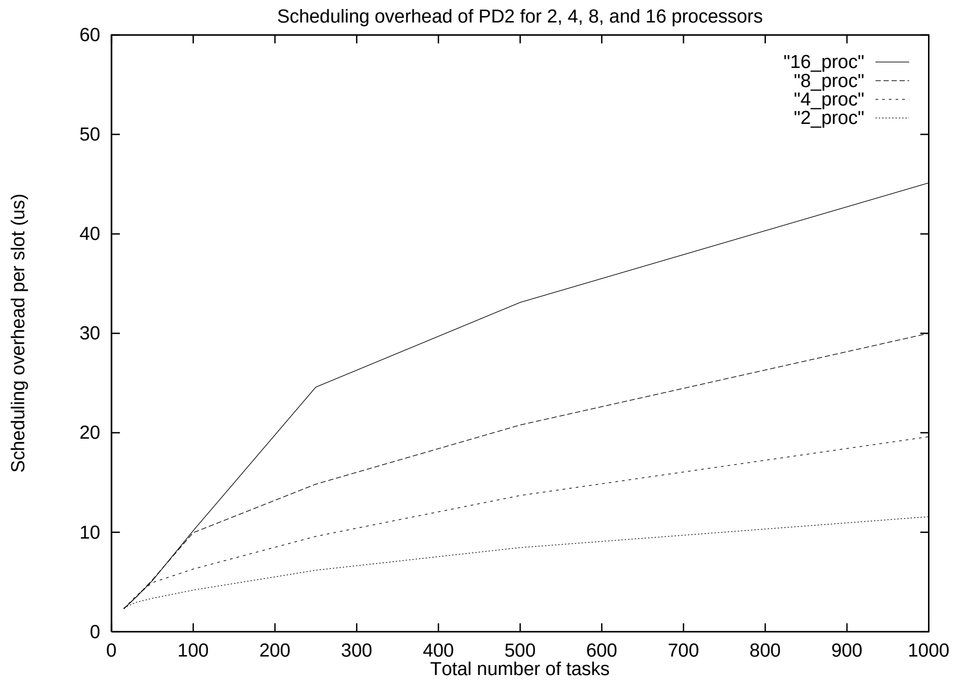



Figure 3.24: Scheduling overhead of PD$^2$ on 2, 4, 8, and 16 processors.

$C(T)$ denote the maximum cache-related preemption delay suffered by task $T$. Let $P_T$ denote the set of tasks that are assigned to the same processor as $T$ and that have periods larger than $T$'s period. Note that $T$ can only preempt tasks in $P_T$. To see why, consider a job $J$ of $T$. Any job $J'$ of $T'$ that is executing prior to the release of $J$ can be preempted by $J$ only if $J'$ has a later deadline. In that case, $T'$ has a larger period than $T$'s, *i.e.*, $T' \in P_T$. Also, job $J$ can preempt at most one task in $P_T$: once $J$ starts executing, no task in $P_T$ will be scheduled by EDF until it completes. Thus, the overhead incurred due to the preemption of any task by $J$ is bounded by $\mathbf{max}_{U \in P_T}\{C(U)\}$, and can be accounted for by inflating $T$'s execution cost by this amount.

Under PD$^2$, the cache-related preemption delay of a task $T$ is the product of $C(T)$ and the worst-case number of preemptions that a job of $T$ can suffer (as derived earlier).

**Simulation experiments.** We now give formulae that show how to inflate task execution costs to include all of the above-mentioned overheads. Let $S_A$ be the scheduling overhead per invocation of scheduling algorithm $A$. Let $c$ denote the cost of a single context switch. Let $P_T$ and $C(T)$ be as defined above. Let $e$ and $p$ be the execution time and period of task $T$. Let $q$ denote the quantum size. (Then, the number of quanta spanned by $t$ time units is $\lceil t/q \rceil$.) The inflated execution cost $e'$ of task $T$ can be computed as follows. (We assume that $p$ is a multiple of $q$.)

$$e' = \begin{cases} e + 2(S_{EDF} + c) + \mathbf{max}_{U \in P_T}\{C(U)\} & \text{, under EDF} \\ e + \left\lceil \frac{e'}{q} \right\rceil \cdot S_{PD^2} + c + \mathbf{min}(\left\lceil \frac{e'}{q} \right\rceil - 1, \frac{p}{q} - \left\lceil \frac{e'}{q} \right\rceil) \cdot (c + C(T)) & \text{, under PD}^2 \end{cases} \tag{3.34}$$

Note that in the formula for PD$^2$, $e'$ occurs on the right-hand side as well because the number of preemptions suffered by a task may vary with its execution cost. $e'$ can be easily computed by initially letting $e' = e$ and by repeatedly applying this formula until the value converges. Simulation experiments conducted by us on randomly-generated task sets suggest that convergence usually occurs within five iterations.

Using the above execution times, the schedulability tests are simple. For PD$^2$, we can use (2.10). For EDF-FF, we invoke the first-fit heuristic to partition the tasks: since the new execution cost of a task depends on tasks on the same processor with larger periods, we consider tasks in the order of decreasing periods.

To measure the schedulability loss caused by both system overheads and partitioning, we conducted a series of simulation experiments. In these experiments, the value of $c$ is fixed at 5 $\mu s$. (As mentioned earlier, $c$ is likely to be between 1 and 10 $\mu s$ in modern processors.) $S_{EDF}$ and $S_{PD^2}$ were chosen based on the values obtained by us in the scheduling-overhead experiments described earlier (refer to Figures 3.23 and 3.24). $C(T)$ was chosen randomly between 0 $\mu s$ and 100 $\mu s$; the mean of this distribution was chosen to be 33.3$\mu s$. We chose the mean of 33.3 $\mu s$ by extrapolating results from [LHS$^+$98, LLH$^+$01]. (Based on cache-cost figures reported by the authors of these papers, we estimated that the maximum cache-related preemption delay would be approximately 30 $\mu s$ in their experiments, assuming a bus speed of 66MHz.) The quantum size of the PD$^2$ scheduler was then assumed to be 1 $ms$.

The number of tasks, $N$, was chosen from the set $\{50, 100, 250, 500, 1000\}$. For each $N$, we generated random task sets with total utilizations ranging from $N/30$ to $\mathbf{min}(N/3, 64)$, *i.e.*, the mean utilization of the tasks was varied from 1/30 to $\mathbf{min}(1/3, 64/N)$. In each step, we generated 1,000 task sets with the selected total utilization. For each task set, we applied (3.34) to account for system overheads and then computed the minimum number of processors required by PD$^2$ and EDF-FF to render the task set schedulable.

Figure 3.25 shows the averages of these numbers for $N = 50$. Note that when the total utilization is at most 3.75, both EDF and PD$^2$ give almost identical performance.

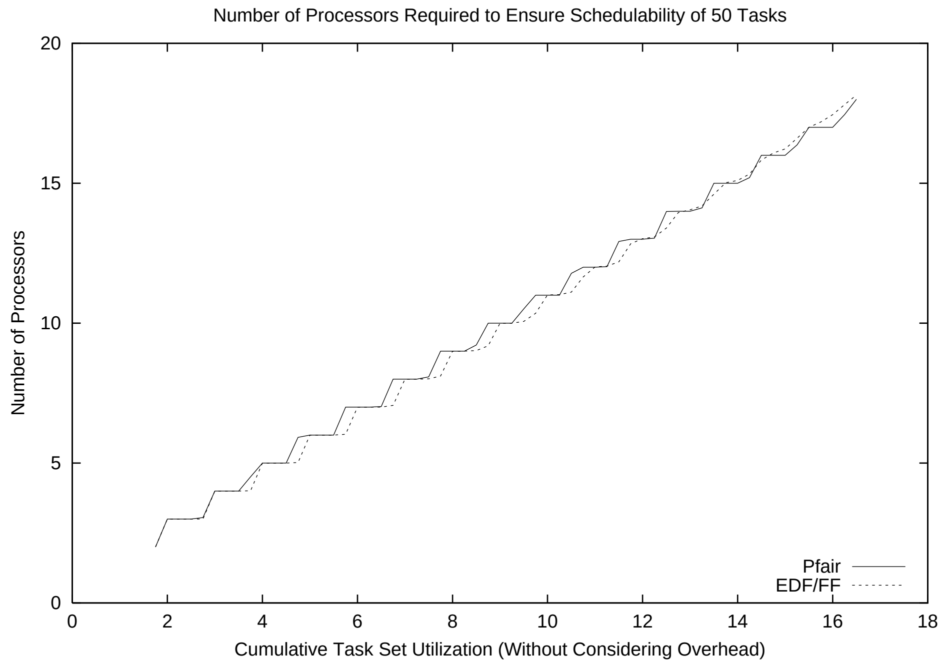



Figure 3.25: Effect of introducing overheads in the schedulability tests. Figure shows the minimum number of processors required by PD$^2$ and EDF for a given total utilization of a system of 50 tasks. As before, 99% confidence intervals were computed for each graph but are not shown because the relative error associated with each point is very small (less than 1% of the reported value).

EDF consistently gives better performance than PD$^2$ in the range $[4, 14)$, after which PD$^2$ gives slightly better performance. This is intuitive because when the utilization of each task is small, the overheads of PD$^2$ and EDF-FF are both negligible. As the utilizations increase, the influence of these overheads is magnified. Though the system overhead of EDF remains low throughout, the schedulability loss due to partitioning grows quickly, as can be seen in Figure 3.26. After a certain point, this schedulability loss overtakes the other overheads. Note that, although EDF does perform better, PD$^2$ is always competitive.

The jagged nature of the lines in the graphs can be explained as follows. Recall that for each randomly-generated task set, we calculate the minimum number of processors required by each scheduling algorithm. For most of the task sets generated with a given total utilization, the number of processors required is identical. Hence, the average is close to an integer. As the total utilization increases, this average increases in spurts of one, resulting in jagged lines.

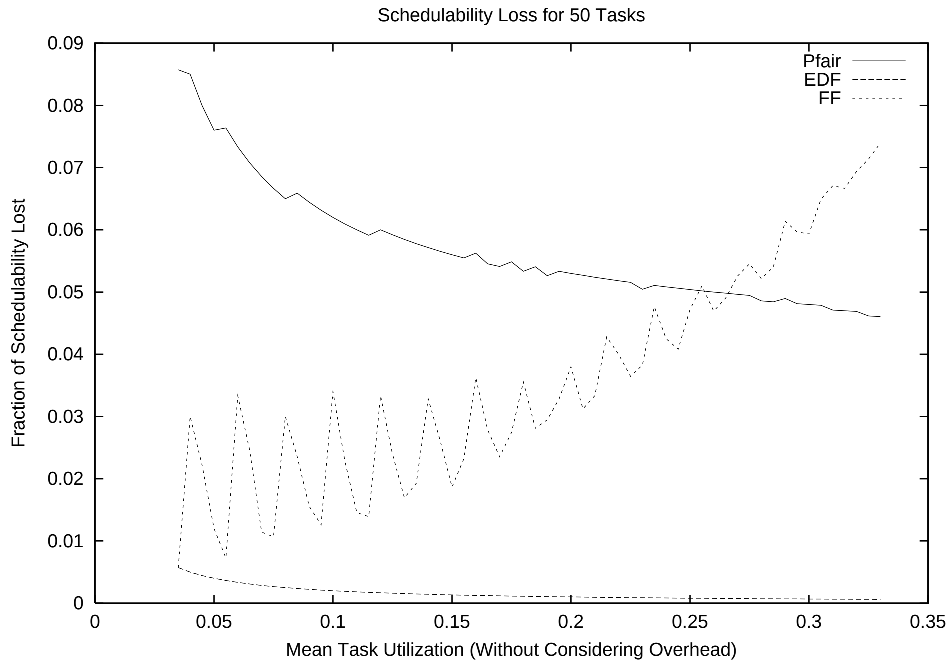
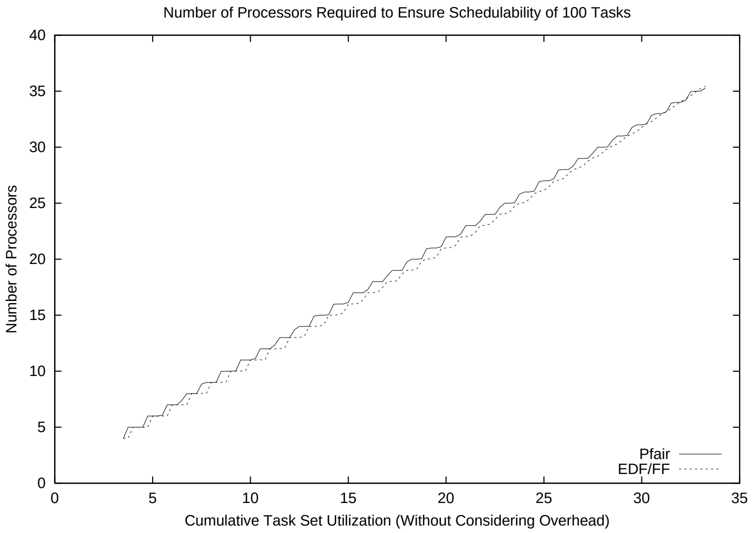



Figure 3.26: Fraction of schedulability loss due to partitioning and due to system overheads under PD$^2$ and EDF-FF for systems of 50 tasks. 99% confidence intervals were computed for each graph but, once again, are not shown. For each EDF and PD$^2$ point sample, relative error is very small (less than 4% and 2%, respectively). The relative error associated with the FF point samples is higher but drops off quickly as utilization increases; the error for these samples is initially approximately 17%, and is below 5% for all mean utilization above 0.22. The variation of adjacent FF sample values appears to be a fairly consistent indicator of the confidence interval's size at each point.

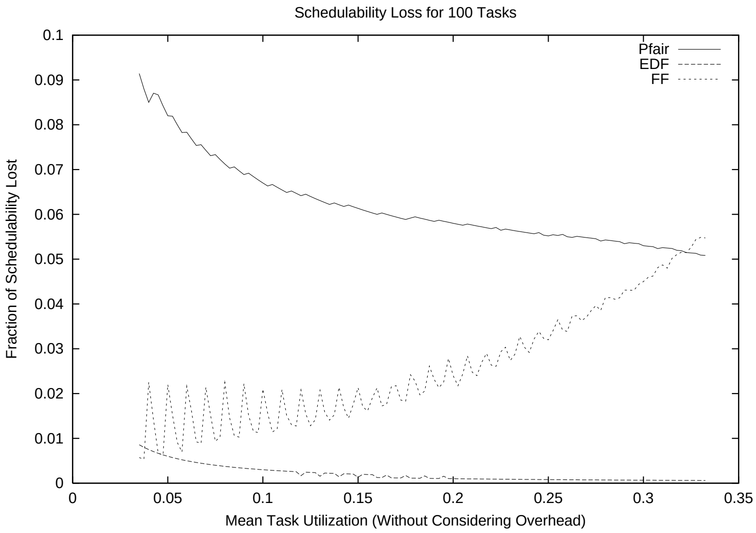
The above trend is seen for other values of $N$ as well. Figures 3.27 and 3.28 show similar graphs for $N = 100$, 250, and 500. Note that, for a given total utilization, mean utilization decreases as the number of tasks increases. As a result, the improvement in the performance of EDF-FF is more than that of PD$^2$. Therefore, the point at which PD$^2$ performs better than EDF-FF occurs at a higher total utilization.

## 3.6 Summary

In this chapter, we proved that the PD$^2$ algorithm is optimal for scheduling asynchronous periodic tasks on multiprocessors. We also performed experiments that show that PD$^2$ is a viable option to partitioning. In spite of the frequent context-switching

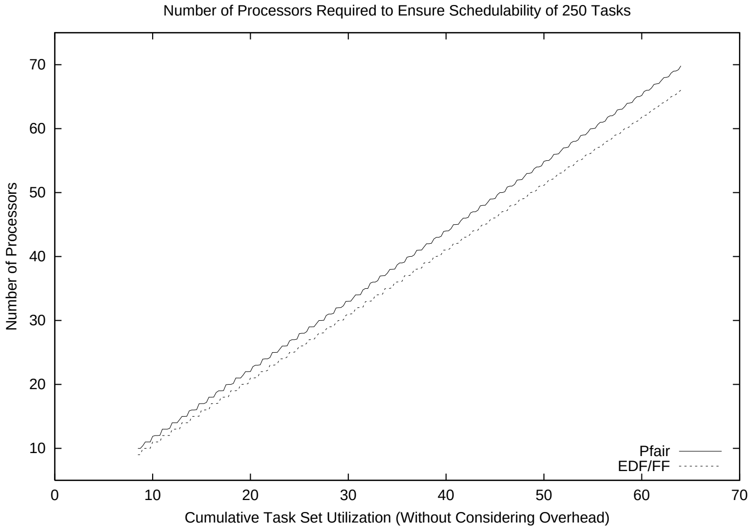



**(a)**

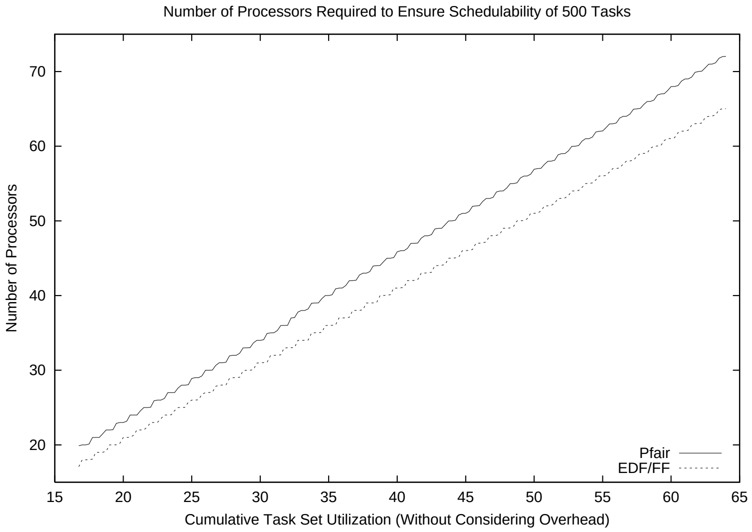



**(b)**

Figure 3.27: **(a)** The minimum number of processors required by PD$^2$ and EDF for a given total utilization of a system of 100 tasks. **(b)** Fraction of schedulability loss due to partitioning and due to system overheads under PD$^2$ and EDF-FF for systems of 100 tasks.




**(a)**




**(b)**

Figure 3.28: The minimum number of processors required by $PD^2$ and EDF for a given total utilization of a system of **(a)** 250 tasks, and **(b)** 500 tasks.

and cache-related overheads, PD$^2$ performs competitively because the schedulability loss due to these overheads is offset by the fact that PD$^2$ provides much better analytical bounds on schedulability than partitioning. Moreover, the results of our experiments are biased against Pfair scheduling in that only *static* systems with *independent* tasks of *low weight* were considered. As we discuss below, it may be more advantageous to use PD$^2$ in several categories of systems.

**Systems with heavy tasks.** Bin-packing heuristics, including those used to assign tasks to processors, usually perform better with smaller items. As the mean task utilization increases, the performance of these heuristics tends to degrade, resulting in more schedulability loss. Although heavy tasks are rare, some techniques, such as hierarchical scheduling [MR99, HA01], can introduce heavy "server" tasks. Hence, it is unrealistic to assume that only light tasks will occur in practice. As our experiments indicate, PD$^2$ performs better than EDF-FF when the mean task utilization is high.

**Dynamic systems.** As described in Section 2.1.3, partitioning approaches are not suitable for dynamic task systems. In addition to the problems mentioned in Section 2.1.3, the dependency of a task's execution requirement on that of other tasks (refer to Equation (3.34)) complicates bin-packing in dynamic task systems. In particular, the execution requirement of a task is affected by tasks with larger periods. Thus, when a new task arrives, checking feasibility also entails re-calculation of the execution requirements of potentially all the tasks with larger periods in the system (and hence, potentially, all the tasks in the system). The associated overhead may be unacceptable in many systems. On the other hand, PD$^2$ does not suffer from this problem because the execution requirements are calculated independently of the other tasks in the system (refer to (3.34)). Further, as we show in the next chapter, dynamic task systems are easier to handle under PD$^2$.

**Resource and object sharing.** As discussed in Section 2.1.3, partitioning is problematic if tasks share resources. Though we do not address the problem of shared resources under Pfair scheduling in this dissertation, Holman and Anderson [HA02b, HA02a] have recently presented efficient synchronization protocols for use with PD$^2$.

The tight synchrony in Pfair scheduling can be exploited to simplify task synchronization. Specifically, each subtask's execution is effectively non-preemptive within its time slot. As a result, problems stemming from the use of locks can be altogether avoided by ensuring that all locks are released before each quantum boundary. The latter is easily accomplished by delaying the start of critical sections that are not guaranteed to complete by the quantum boundary. When critical-section durations are short compared to the quantum length, which is expected to be the common case, this approach can be used to provide synchronization with very little overhead[HA02a]. (In experiments conducted by Ramamurthy [Ram97] on a 66 MHz processor, critical-section durations for a variety of common objects (*e.g.*, queues, linked lists, *etc.*) were found to be in the range of tens of microseconds. On modern processors, these operations will likely require no more than a few microseconds. Note that, in a Pfair-scheduled system, if such a critical section were preempted, then its length could easily be increased by several orders of magnitude.)

The tight synchrony in Pfair scheduling also facilitates the use of *lock-free* shared objects. Operations on lock-free objects are usually implemented using "retry loops." Lock-free objects are of interest because they do not give rise to priority inversions and can be implemented with minimal operating system support. Implementation of such objects in multiprocessor systems has been viewed as being impractical because deducing bounds on retries due to interferences *across* processors is difficult. However, Holman and Anderson have shown that the tight synchrony in Pfair-scheduled systems can be exploited to obtain reasonably tight bounds on multiprocessors [HA02b].

# Chapter 4

# Rate-based Scheduling*

In many systems, task invocations may not match the periodic or sporadic pattern of invocations. For example, consider a teleconferencing application that is receiving video frames over a network. Even if the frames are sent periodically, their arrival times may not be periodic or even sporadic. In fact, the arrivals can be bursty or late due to network congestion and other factors. Jeffay *et al.* [JB95, JG99] presented the uniprocessor *rate-based execution* (*RBE*) model as a way to handle this scenario. In the RBE model, each task has four parameters: $x$, $y$, $d$, and $e$. Such a task releases on an average $x$ jobs every $y$ time units; each job requires $e$ units of processor time and must be completed within $d$ time units of its release. However, the instantaneous rate of job releases is allowed to be more; scheduling of early jobs is handled by an appropriate postponement of their deadlines.

In order to handle similar scenarios in multiprocessor systems, we introduce the *intra-sporadic* (IS) task model as an extension of the sporadic model. We formally define it in the next section. In Sections 4.2 and 4.3, we consider task systems in which tasks are allowed to join and leave the system. We also derive sufficient conditions under

*The results presented in this chapter have been published in the following papers.

[AS00b] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, December 2000.

[SA02] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[SA04b] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 2004. Under submission. (A preliminary version of this paper was presented at the 11th International Workshop on Parallel and Distributed Real-time Systems.)

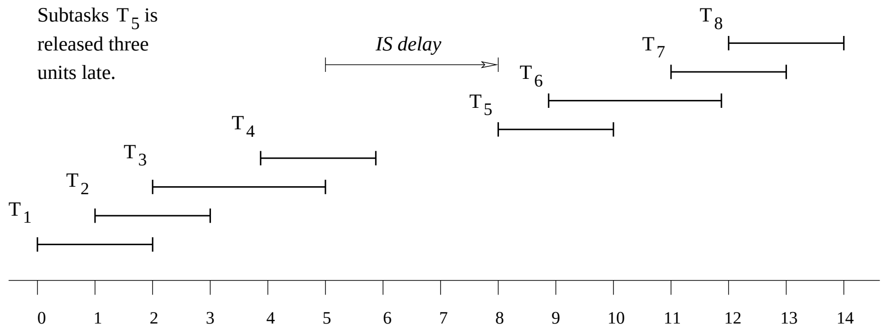



Figure 4.1: The PF-windows of the first eight subtasks of an IS task $T$ with weight 8/11. Subtask $T_5$ is released three units late causing all later subtask releases to be delayed by three time units.

which joins and leaves can occur in PD$^2$-scheduled systems without causing any missed deadlines; we present counterexamples demonstrating the tightness of these conditions. As a corollary of this main result, it follows that PD$^2$ is optimal for scheduling IS task systems (and hence, sporadic task systems as well).

## 4.1 The Intra-sporadic Task Model

Recall that the sporadic model generalizes the periodic model by allowing jobs to be released late. The IS model generalizes this notion further by allowing *subtasks* to be released late, as illustrated in Figure 4.1. More specifically, the separation between subtask releases $r(T_i)$ and $r(T_{i+1})$ is allowed to be more than $\lfloor i/wt(T) \rfloor - \lfloor (i-1)/wt(T) \rfloor$, which would be the separation if $T$ were periodic (refer to Equation 2.7). Thus, an IS task is obtained by allowing a task's windows to be right-shifted from where they would appear if the task were periodic. Figure 4.1 illustrates this. Each subtask of an IS task has an *offset* that gives the amount by which its window has been right-shifted. The offset of subtask $T_i$ is denoted $\theta(T_i)$. By (2.7) and (2.8), we have the following.

$$r(T_i) = \theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{4.1}$$

$$d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil \tag{4.2}$$

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

$$k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i) \tag{4.3}$$

Because $\left\lfloor \frac{i}{wt(T)} \right\rfloor \geq \left\lceil \frac{i}{wt(T)} \right\rceil - 1$, by (4.1), $r(T_{i+1}) \geq \theta(T_{i+1}) + \left\lceil \frac{i}{wt(T)} \right\rceil - 1$. Hence, by (4.2) and (4.3), it follows that

$$r(T_{i+1}) \geq d(T_i) - 1. \tag{4.4}$$

Each subtask $T_i$ has an additional parameter $e(T_i)$ that specifies the first time slot at which it is eligible to be scheduled. It is assumed that $e(T_i) \leq r(T_i)$ and $e(T_i) \leq e(T_{i+1})$ for all $i \geq 1$. Allowing $e(T_i)$ to be less than $r(T_i)$ is equivalent to allowing "early" subtask releases as in ERfair scheduling. We refer to the interval $[r(T_i), d(T_i))$ as the *PF-window* of $T_i$ and the interval $[e(T_i), d(T_i))$ as its *IS-window*. The inequality (4.4) implies that PF-windows of consecutive subtasks of a task overlap by at most one slot.

The validity of a schedule for an IS task system is given by the definition below.

**Definition 4.1.** *A valid schedule for an IS task system is one that satisfies the following properties:* **(i)** *each subtask is scheduled in its IS-window,* **(ii)** *two subtasks of the same task are not scheduled in the same slot, and* **(iii)** *the number of subtasks scheduled in any slot is at most the number of processors.* ■

Since $e(T_i) \leq r(T_i)$, a subtask's PF-window is contained in its IS-window. Later, when obtaining the feasibility condition for IS task systems, we use this fact to obtain a valid schedule for a feasible IS task system.

Note that the notion of a job is secondary to the notion of a subtask in IS task systems. For systems in which subtasks are grouped into jobs that are released in sequence, the definition of $e$ would preclude a subtask from becoming eligible before the beginning of its job. Using the definitions above, it is easy to show that sporadic and periodic tasks are special cases of IS tasks. In particular, a sporadic task $T$ is an IS task in which only the first subtask of each job may be released late, *i.e.*, if $T_i$ and $T_{i+1}$ are part of the same job, then $\theta(T_i) = \theta(T_{i+1})$. A periodic task $T$ is an IS task such that only the very first subtask of each task may be released late, *i.e.*, $\theta(T_i) = \theta(T_1)$ for all $i \geq 1$. (For a synchronous periodic task, $\theta(T_1) = 0$ holds.) Note that, by defining the function $e$ appropriately, we can obtain eligibility intervals (*i.e.*, IS-windows) like those in either a Pfair or ERfair system. In fact, we can define eligibility intervals

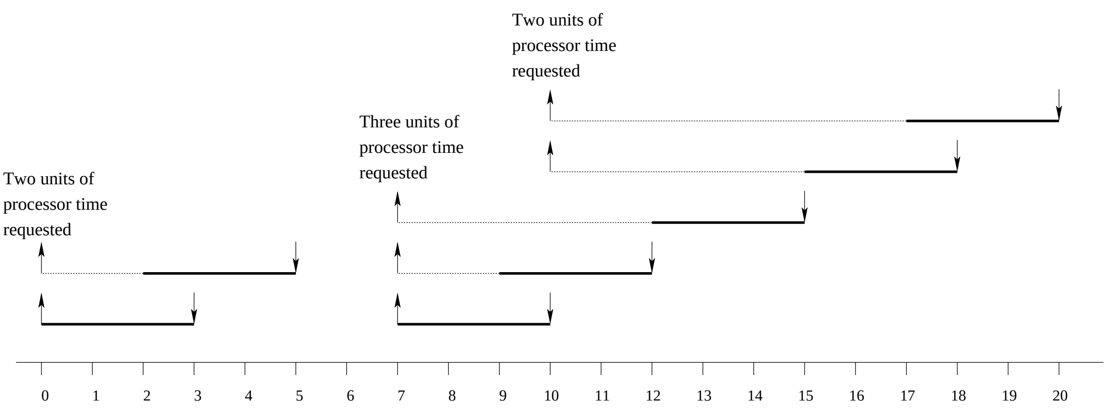



Figure 4.2: The up arrows corresponds to subtask eligibility times and down arrows correspond to subtask deadlines. The dotted lines are used to illustrate eligibility windows and the bold lines are used to illustrate the PF-windows. A server with weight 2/5 is shown. It receives requests of two units of processor time at times 0 and 10, and a request of three units of processor time at time 7.

(*i.e.*, IS-windows) that are longer than in a Pfair system but shorter than in an ERfair system.

**Usefulness of the IS task model.** Figure 4.2 illustrates an example server that reserves a processor share of 2/5 (given by its weight) and receives requests requiring 2 or 3 units of processor time. Thus, the IS model allows us to decouple the request size from the service rate.

The IS model also allows the instantaneous rate of subtask releases to differ greatly from the average rate (given by a task's weight). Hence, it is more suitable than the periodic model for several applications in networking. Examples include web servers that provide quality-of-service guarantees, packet scheduling in networks, and the scheduling of packet-processing activities in routers [SHA+03]. Due to network congestion and other factors, packets may arrive late or in bursts. The IS model treats these possibilities as first-class concepts and handles them more seamlessly. In particular, a late packet arrival corresponds to an IS delay. On the other hand, if a packet arrives early (as part of a bursty sequence), then its eligibility time will be less than its Pfair release time. Note that its Pfair release time determines its deadline. Thus, in effect, an early packet arrival is handled by postponing its deadline to where it would have been had the packet arrived on time.

**Feasibility.** We now prove that the following expression is a feasibility condition for an IS task system $\tau$ on $M$ processors.

$$\sum_{T \in \tau} wt(T) \leq M \tag{4.5}$$

Let $t_l$ be an arbitrary time slot. We show that $\tau$ has a valid schedule over the time interval $[0, t_l)$ by considering flows in a certain graph $G(\tau, t_l)$. By examining the windows of all the subtasks that have deadlines in the interval $(0, t_l]$, we construct a flow graph $G(\tau, t_l)$ such that a maximal flow $f$ in $G(\tau, t_l)$ corresponds to a valid schedule for these subtasks. As mentioned earlier, we construct a valid schedule in which each subtask is scheduled in its PF-window (which implies that it is scheduled in its IS-window).

**Definition of $G(\tau, t_l)$.** Let $ns(T, t_l)$ denote the number of subtasks of $T$ that have deadlines in the interval $(0, t_l]$, and let $et(T, t_l)$ denote the union of the PF-windows of the first $ns(T, t_l)$ subtasks of $T$.

The vertex set $V$ of $G(\tau, t_l)$ is the union of six disjoint sets of vertices $V_0, \ldots, V_5$ and the edge set $E$ is the union of five disjoint sets of weighted edges $E_0, \ldots, E_4$, where $E_i$ is a subset of $V_i \times V_{i+1} \times \mathbf{N}^+$, $0 \leq i \leq 4$. Thus, $G$ is a six-layered graph, with all the edges connecting vertices in adjacent layers. The vertex sets $V_0, \ldots, V_5$ are defined as follows.

$V_0 = \{\text{source}\}$.
$V_1 = \{\langle 1, T\rangle \mid T \in \tau\}$, corresponding to tasks.
$V_2 = \{\langle 2, T, i\rangle \mid T \in \tau,\ 1 \leq i \leq ns(T, t_l)\}$, corresponding to subtasks.
$V_3 = \{\langle 3, T, t\rangle \mid T \in \tau,\ t \in et(T, t_l)\}$, corresponding to subtask windows.
$V_4 = \{\langle 4, t\rangle \mid 0 \leq t \leq t_l\}$, corresponding to time.
$V_5 = \{\text{sink}\}$

The edge sets $E_0, \ldots, E_4$ are defined as follows.

$E_0 = \{(\text{source}, \langle 1, T\rangle, ns(T, t_l)) \mid T \in \tau\}$
$E_1 = \{(\langle 1, T\rangle, \langle 2, T, i\rangle, 1) \mid T \in \tau,\ 1 \leq i \leq ns(T, t_l)\}$
$E_2 = \{(\langle 2, T, i\rangle, \langle 3, T, t\rangle, 1) \mid T \in \tau,\ 1 \leq i \leq ns(T, t_l),\ t \in w(T_i)\}$
$E_3 = \{(\langle 3, T, t\rangle, \langle 4, t\rangle, 1) \mid T \in \tau,\ t \in et(T, t_l)\}$
$E_4 = \{(\langle 4, t\rangle, \text{sink}, M) \mid 0 \leq t \leq t_l\}$

The weight of an edge in $E_0$ corresponds to the total processing time required by a task in the interval $[0, t_l)$. The edges in $E_1$ are used to add the restriction that tasks are allocated in terms of subtasks (*i.e.*, quanta). The edges in $E_2$, $E_3$, and $E_4$ are used to ensure the validity of the resulting schedule (refer to Definition 4.1

A flow is called *integral* if and only if flow across each edge is integral. We use the following theorem about integral flows in graphs with integral edge capacities.

**Theorem 4.1 (Ford and Fulkerson [FF62]).** *A graph in which all edge capacities are integral has a integral maximal flow.*

Figure 4.3 shows an example graph for a set of tasks $T$, $U$, $V$, and $W$, where $wt(T) = 3/7$, $wt(U) = 1/6$, $wt(V) = 4/7$, and $wt(W) = 5/6$. Note that these tasks fully utilize two processors. The time interval is $[0, 8)$. In this interval, all subtasks are released as in a periodic task system, except that the second subtask of $T$ is released one time unit late. The flow shown in Figure 4.3 corresponds to the following schedule: $W_1$ and $V_1$ are scheduled in slot 0, $W_2$ and $T_1$ in slot 1, $W_3$ and $V_2$ in slot 2, and $W_4$ and $V_3$ in slot 3.

**Feasibility proof.** The existence of a schedule for an IS task system $\tau$ that satisfies Expression (4.5) follows from Lemmas 4.1 and 4.2 below.

**Lemma 4.1.** *If there exists an integral flow of size* $\sum_{T \in \tau} ns(T, t_l)$ *in* $G(\tau, t_l)$*, then there exists a valid schedule for* $\tau$ *over the interval* $[0, t_l)$.

*Proof.* An integral flow of size $\sum_{T \in \tau} ns(T, t_l)$ implies that the flow out of the source is $\sum_{T \in \tau} ns(T, t_l)$. By definition of $E_0$, the sum of the capacities of all the outgoing edges from the source is $\sum_{T \in \tau} ns(T, t_l)$. Therefore, all the edges in $E_0$ carry a flow equal to their capacity. Hence, an edge from the source to $\langle 1, T \rangle \in V_1$ carries a flow equal to $ns(T, t_l)$. Because there are $ns(T, t_l)$ outgoing edges from $\langle 1, T \rangle$, and each edge of $E_1$ has a capacity of 1, each such edge carries a flow of 1. Therefore, the flow into each vertex in $V_2$ is 1.

We obtain a schedule for $\tau$ by scheduling $T_i$ in slot $t$ if and only if there is a flow of 1 from vertex $\langle 2, T, i \rangle$ to $\langle 3, T, t \rangle$. From the following discussion and by Definition 4.1, it follows that this schedule is valid.

**(i)** Since each outgoing edge from $V_2$ has a capacity of 1, and because the flow is integral, this implies that at most one edge, starting from any vertex in $V_2$, has a non-zero flow. Thus, for each subtask $T_i$, only one of $\langle 3, T, t \rangle$ has an incoming flow of 1. In other

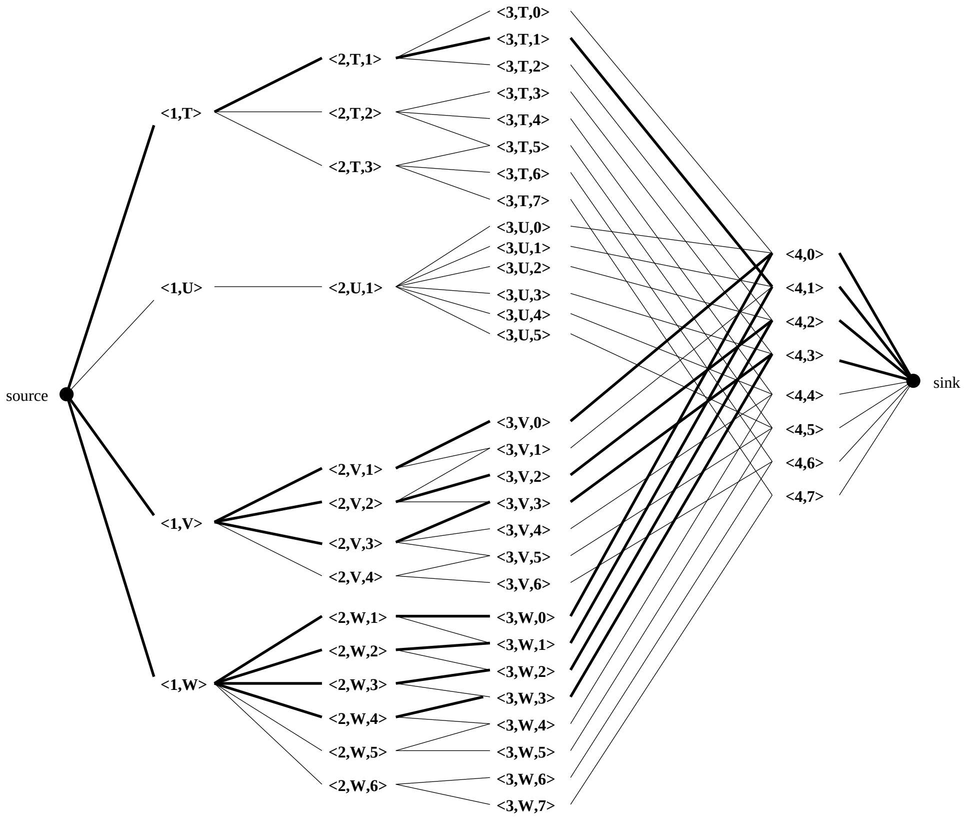



Figure 4.3: Example flow graph for a task set consisting of four tasks $T$, $U$, $V$, and $W$, where $wt(T) = 3/7$, $wt(U) = 1/6$, $wt(V) = 4/7$ and $wt(W) = 5/6$. Subtask $T_2$ is released one slot late. The bold lines in the figure correspond to a flow $f$. The value of $f$ is 8, which indicates that eight subtasks have been scheduled.

words, a subtask is scheduled at most once. Because there is an edge from $\langle 2, T, i\rangle$ to $\langle 3, T, t\rangle$ only if $t$ lies in $T_i$'s PF-window, $T_i$ is scheduled in its PF-window.

**(ii)** Note that, because successive PF-windows of the same task may overlap by one slot, vertex $\langle 3, T, t\rangle$ can have more than one incoming edge. However, because the edge from $\langle 3, T, t\rangle$ to $\langle 4, t\rangle$ has a capacity of 1, at most one such incoming edge can have a flow of 1. This ensures that multiple subtasks of the same task are not scheduled in the same slot.

**(iii)** Because each edge in $E_4$ has a capacity of $M$, there can be at most $M$ edges in $E_3$ with a flow of 1 that are incident on the same vertex in $V_4$. In other words, at most $M$ subtasks are scheduled in a single slot. ■

Note that the maximum flow of $G(\tau, t_l)$ is at most $\sum_{T \in \tau} ns(T, t_l)$, because this is the sum of the capacities of all edges coming from the source. We now show that a real-valued flow of such a size exists.

**Lemma 4.2.** *$G(\tau, t_l)$ has a real-valued flow of size $\sum_{T \in \tau} ns(T, t_l)$.*

*Proof.* We use the following flow assignments. These assignments are similar to those given by Baruah *et al.* [BCPV96] to establish that Expression (2.10) is a feasibility condition for synchronous, periodic tasks.

- Each edge (source, $\langle 1, T\rangle, ns(T, t_l)) \in E_0$ carries a flow of size $ns(T, t_l)$.
- Each edge $(\langle 1, T\rangle, \langle 2, T, i\rangle, 1) \in E_1$ carries a flow of 1. Because there are $ns(T, t_l)$ outgoing edges from each $\langle 1, T\rangle$, flow is conserved at all vertices in $V_1$.
- The flow through the edges in $E_2$ is defined as follows. Let $f(T_i, t)$ define the flow from $\langle 2, T, i\rangle$ to $\langle 3, T, t\rangle$. Then,

$$f(T_i, u) = \begin{cases} \left(\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1\right) \times wt(T) - (i-1), & u = r(T_i) \\ i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T), & u = d(T_i) - 1 \\ wt(T), & r(T_i) + 1 \le u \le d(T_i) - 2 \\ 0, & \text{otherwise.} \end{cases} \tag{4.6}$$

  We now show that these assignments ensure that the flow is conserved at every vertex in $V_2$, *i.e.*, the flow out of each vertex $\langle 2, T, i\rangle \in V_2$ is 1. By (4.6), the total flow out of $\langle 2, T, i\rangle$ is $(d(T_i) - r(T_i) - 2) \times wt(T) + \left(\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1\right) \times wt(T) - (i-1)$ $+ i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T)$, which simplifies to $1 + wt(T) \times (d(T_i) - r(T_i)) +$ $wt(T) \times \left(\left\lfloor \frac{i-1}{wt(T)} \right\rfloor - \left\lceil \frac{i}{wt(T)} \right\rceil\right)$. By (4.1) and (4.2), $d(T_i) - r(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil -$ $\left\lfloor \frac{i-1}{wt(T)} \right\rfloor$. Thus, the total flow is 1.
- Each edge $(\langle 3, T, t\rangle, \langle 4, t\rangle, 1) \in E_3$ carries a flow equal to the sum of all incoming flows at $\langle 3, T, t\rangle$. We now show that this flow is at most $wt(T)$ (which is at most 1). We first show that $f(T_i, t) \le wt(T)$. This follows directly from (4.6) if $t \notin \{r(T_i), d(T_i - 1)\}$. If $t = r(T_i)$, then $f(T_i)$ is

$$\left(\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1\right) \times wt(T) - (i-1) \quad , \quad \text{by (4.6)}$$

$$
\begin{aligned}
&\le \left(\frac{i-1}{wt(T)}+1\right) \times wt(T) - (i-1) &&, \ \lfloor x \rfloor \le x \\
&= wt(T) &&, \ \text{by simplification}
\end{aligned}
$$

If $t = d(T_i) - 1$, then $f(T_i)$ is

$$
\begin{aligned}
& i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T) &&, \ \text{by (4.6)} \\
\le\ & i - \left(\frac{i}{wt(T)} - 1\right) \times wt(T) &&, \ \lceil x \rceil \ge x \Rightarrow -\lceil x \rceil \le -x \\
=\ & wt(T) &&, \ \text{by simplification}
\end{aligned}
$$

We now only need to consider the time slot in which two consecutive PF-windows overlap. That will be the case when $d(T_i) - 1 = r(T_{i+1})$ for some $i$. In this case, the total flow will be $f(T_i, d(T_i) - 1) + f(T_{i+1}, r(T_{i+1}))$. Thus, the flow is $i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T) + \left(\left\lfloor \frac{i}{wt(T)} \right\rfloor + 1\right) \times wt(T) - i$, which simplifies to $\left(\left\lfloor \frac{i}{wt(T)} \right\rfloor - \left\lceil \frac{i}{wt(T)} \right\rceil + 2\right) \times wt(T)$. Since, $d(T_i) - 1 = r(T_{i+1})$, it follows that $\theta(T_i) = \theta(T_{i+1})$ and $\left\lceil \frac{i}{wt(T)} \right\rceil - 1 = \left\lfloor \frac{i}{wt(T)} \right\rfloor$. Therefore, $\left\lfloor \frac{i}{wt(T)} \right\rfloor - \left\lceil \frac{i}{wt(T)} \right\rceil = -1$. Thus, the total flow is $wt(T)$. Thus, in all cases, the sum of all incoming flows at $\langle 3, T, t\rangle$ is at most $wt(T)$.

- Each edge $(\langle 4, t\rangle, \text{sink}, M) \in E_4$ carries a flow equal to the sum of all incoming flows at $\langle 4, t\rangle$. Thus, the incoming flow into $\langle 4, t\rangle$ from $\langle 3, T, t\rangle$ can be at most $\sum_{T\in\tau} wt(T)$. Because $\sum_{T\in\tau} wt(T) \le M$, the incoming flow and hence the outgoing flow at $\langle 4, t\rangle$ is at most $M$.

This proves that the flow along each edge is at most its capacity and that the flow is conserved at all vertices. Hence, the flow defined above is a valid flow. ■

We are now in a position to state the following lemma and theorem.

**Lemma 4.3.** *An IS task system $\tau$ has a valid schedule on $M$ processors in which each subtask is scheduled in its PF-window if and only if $\sum_{T\in\tau} wt(T) \le M$.*

*Proof.* The necessity of the condition follows from the necessity of condition for periodic task systems (refer to Theorem 2.8) since any periodic task system is also an IS task system. To prove sufficiency, we construct a valid schedule for $\tau$ in $[0, t)$ for any given $t$.

Because $t$ is arbitrary, it follows that $\tau$ has a valid schedule. By Lemma 4.2, it follows that $G(\tau, t)$ has a real-valued flow of size $\sum_{T\in\tau} ns(T,t)$. This is a maximal flow, since the sum of the capacities of all the outgoing edges from the source is the same. Because all edge capacities in $G(\tau, t)$ are integers, by Theorem 4.1, it follows that $G(\tau, t)$ also has a integral flow of size $\sum_{T\in\tau} ns(T,t)$. Therefore, by Lemma 4.1, $\tau$ has a valid schedule over $[0, t)$. By the definition of $G(\tau, t)$, it follows that in this schedule each subtask will be scheduled in its PF-window. ■

**Theorem 4.2.** *An IS task system $\tau$ has a valid schedule on $M$ processors if and only if $\sum_{T\in\tau} wt(T) \leq M$.*

*Proof.* Follows directly from Lemma 4.3, since every subtask's PF-window is contained in its IS-window. ■

**The PD$^2$ priority definition for IS tasks.** The $b$-bit and group deadline for a subtask are defined as before assuming that future subtask releases are as early as possible. Thus, $b(T_i)$ is given by (2.11), and $D(T_i)$ is given by the following formula (refer to (3.1).

$$D(T_i) = \theta(T_i) + \left\lceil \frac{\left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil \times (1 - wt(T)) \right\rceil}{1 - wt(T)} \right\rceil \tag{4.7}$$

**Necessity of new proof techniques for IS task systems.** Because periodic job releases represent a "worst-case" scenario for an IS task, one may think that the optimality of PD$^2$ for IS tasks follows as a simple corollary from previous work. However, the swapping technique used in the optimality proof for PD$^2$ (presented earlier in Section 3.4) relies *crucially* on the fact that at any moment of time, future window alignments can be predicted. However, such predictions cannot be made for IS task systems. In particular, consider $r(T_{i+1})$ in that proof. We proved that $b(T_i)$ and used this to show that $r(T_{i+1}) = t'$ (refer to (3.12)). However, this inference cannot be made if $T$ were an IS task because $T_{i+1}$ may have been released late. Thus, the whole proof falls apart. Nevertheless, the swapping proof reveals several of the underlying concepts why PD$^2$ works, and also provides invaluable insight into Pfair scheduling in general. (This is the primary reason for presenting the proof even though the optimality of PD$^2$ for IS task systems implies its optimality for asynchronous periodic task systems.)

Another approach that has been often used with uniprocessor scheduling algorithms is to reduce sporadic systems to asynchronous, periodic systems in the following way.

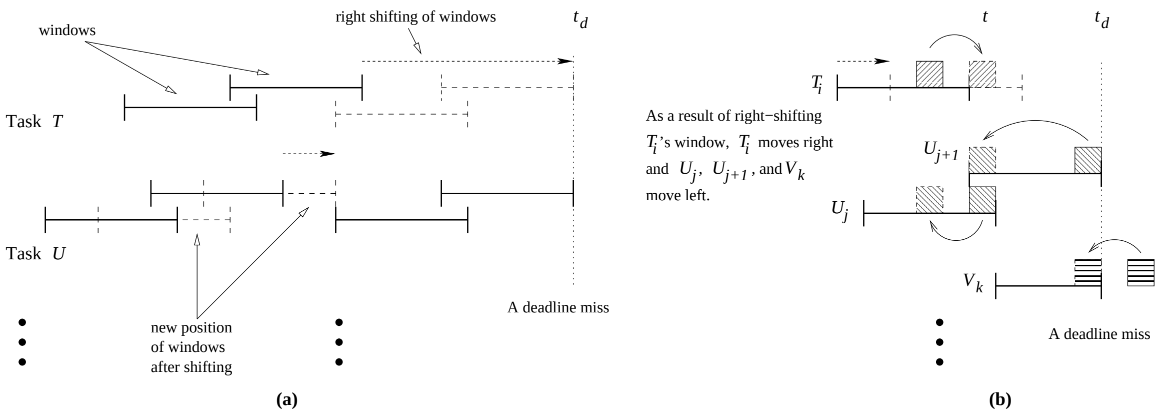



Figure 4.4: **(a)** Starting with a sporadic task set that misses a deadline at $t_d$, we can right-shift all windows towards $t_d$. Intuitively, we *should* get an asynchronous, periodic task system that misses a deadline at $t_d$. **(b)** Unfortunately, right-shifting a window need not increase future demand. Here, shifting $T_i$ to the right by two slots decreases its priority and hence subtask $U_j$, which was scheduled later at time $t$ in the original schedule, may now have higher priority. Note that at time $t$, a processor might be idle, in which case $U_{j+1}$ can now be scheduled in that slot. This can cause a cascade of future left-shifts. Thus, right-shifting a window can in fact *decrease* future demand.

Consider a scheduling algorithm $A$ that has been shown to be correct for asynchronous, periodic tasks. Suppose that there exists a feasible sporadic task system $\tau$ that misses a deadline at some time $t_d$ when scheduled using $A$. Let $S$ be the corresponding schedule. We may assume that all jobs in $S$ after $t_d$ are released in a periodic fashion, because such jobs have no impact on the deadline miss at time $t_d$. Now, if we inductively "right-shift" all jobs released before time $t_d$ in $S$ until there are no sporadic separations among jobs before $t_d$, then we get a schedule $S'$ that is in accordance with the asynchronous, periodic task model (see Figure 4.4(a)). Moreover, "right-shifting" such jobs in $S$ can only increase demand near time $t_d$. Thus, a deadline must be missed at time $t_d$, a contradiction.

To see why this argument cannot be applied in Pfair-scheduled multiprocessor systems, consider the situation shown in Figure 4.4(b). Here, subtask $T_i$ is right-shifted into slot $t$. Before the shift, subtask $U_j$ was scheduled at $t$ and some processor was idle at $t$. After the shift, $U_j$ has higher priority than $T_i$, so the two are swapped in the schedule as a result of shifting $T_i$. Note that $U_j$ being scheduled at $t$ makes $U_{j+1}$ ineligible at $t$. However, after $T_i$ and $U_j$ are swapped, $U_{j+1}$ *is* eligible at $t$ and thus it may left-shift into slot $t$. This may cause a cascade of other left-shifts, which in turn can cause a presumed (future) missed deadline to be met. The root of the problem here is that right-shifting certain subtasks may in fact *reduce* demand in the future.

In Section 4.3, we present a lag-based proof to show the optimality of PD$^2$ for IS task systems. Our proof considers scheduling of dynamic task systems (defined in the following section); the optimality of PD$^2$ for IS task systems follows as a corollary of the main result.

## 4.2 Scheduling of Dynamic Task Systems

In many real-time systems, the set of runnable tasks may change dynamically. One example of such a system is a real-time virtual-reality application in which the user moves within a virtual environment. As the user moves and the virtual scene changes, the time required to render the scene may vary substantially. If a single task is responsible for rendering, then its weight may change frequently. Task *reweighting* can be modeled as a leave-and-join problem, in which a task with the old weight leaves and a task with the new weight joins. Other examples of dynamic systems include embedded systems that support different modes of operation (a mode change may require adding new tasks and deleting existing tasks) and desktop systems that support real-time applications such as multimedia and collaborative-support systems (which may be initiated at arbitrary times).

In dynamic real-time task systems, a key issue is that of devising conditions under which tasks may join and leave the system. A condition for joining is an immediate consequence of the feasibility test in (4.5), *i.e.*, admit a task if the total utilization is at most $M$ after its admission. The important question left is: *when should a task be allowed to leave the system*? Here, we are referring to the time when the task's utilization can be reclaimed. The task may actually be allowed to leave the system earlier. As the following example illustrates, leaves cannot be unrestricted or deadlines may be missed.

Consider a task system consisting of one task of weight $\frac{1}{2}$ and $k \geq 2$ tasks of weight $\frac{1}{2k}$ to be EPDF-scheduled on one processor. At time 0, the task of weight $\frac{1}{2}$ is scheduled. Suppose that it leaves after completing its job and re-joins immediately. Again, at time 1, it will be assigned higher priority and hence, will be scheduled again. Repeating this $2k-2$ times gives us the schedule shown in Figure 4.5. The task of weight $\frac{1}{2}$ effectively executes at a rate of $\frac{2k-1}{2k}$ over the interval $[0, 2k)$, and $k-1$ of the tasks of weight $\frac{1}{2k}$ miss their deadlines at time $2k$. This implies that at least one subtask completes $k-1$

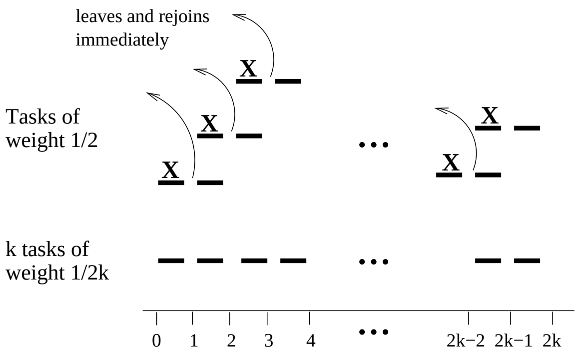



Figure 4.5: A deadline-based schedule for a dynamic task set. $k$ tasks of weight $\frac{1}{2k}$ and one task of weight $\frac{1}{2}$ join at time 0. The PF-windows of all the subtasks are shown. The task of weight $\frac{1}{2}$ is scheduled in $[0, 1)$. (The time slot in which it is scheduled is denoted by an 'X'.) It leaves at time 1 and re-joins immediately. At time 1, it is scheduled again. This pattern repeats until time $2k - 2$ and leads to deadline misses for $k - 1$ of the tasks with weight $\frac{1}{2k}$.

time units after its deadline. Thus, tardiness cannot be bounded by any constant value if leaves are unrestricted. (Note that this example applies to any deadline-based Pfair algorithm such as PF, PD, or PD$^2$.)

The results of Stoica *et al.* [SAWJ$^+$96] and Baruah *et al.* [BGP$^+$97] imply that the following conditions are sufficient on uniprocessor systems to ensure that no task misses its deadline.

**(J0)** *Join condition*: A task $T$ can join at time $t$ if and only if the total utilization after joining is at most one, *i.e.*, one.

**(L0)** *Leave condition*: A task $T$ can leave at time $t$ if and only if $lag(T, t) \geq 0$.

In (J0) and the subsequent join conditions, we assume the following. If a task $T$ joins at time $t$, then $\theta(T_1)$ is set to $t$. A task that re-joins after having left is viewed as a new task.

An IS task requests execution in terms of subtasks. Therefore, if $T_i$ is the last subtask of $T$ that was scheduled, then $lag(T, t) < 0$ holds until time $d(T_i)$. To see why, note that task $T$ receives $i$ units of processor time in the ideal schedule only by time $d(T_i)$. Thus, $t \geq d(T_i)$ is equivalent to $lag(T, t) \geq 0$. Hence, we can re-state the above conditions as follows.

**(J1)** *Join condition*: A task $T$ can join at time $t$ if and only if the total utilization after joining is at most the number of processors.

**(L1)** *Leave condition*: A task $T$ can leave at time $t$ if and only if $t \geq d(T_i)$, where $T_i$ is the last-scheduled subtask of $T$.

Intuitively, if a task is over-allocated, then some other task is under-allocated. Hence, if we allow an over-allocated task to leave, then it can re-join immediately and execute at a higher rate causing an under-allocated task to miss its deadline. Further, in the feasibility proof for IS task systems, graph $G(\tau, t_l)$ has a real-valued flow equal to the number of subtasks if and only if every subtask's PF-window lies in $[0, t_l)$. This is true because no task leaves before the end of the PF-window of its last-scheduled subtask (refer to (L1)). Therefore, a task system that satisfies (J1) and (L1) is feasible on multiprocessors. However, as we show below, (J1) and (L1) are not sufficient on multiprocessors, when any priority-based algorithm such as PF, PD, or PD$^2$ is used.

The theorem below applies to any "weight-consistent" Pfair scheduling algorithm. An algorithm is *weight-consistent* if, given two tasks $T$ and $U$ of equal weight with eligible subtasks $T_i$ and $U_j$, respectively, where $i = j$ and $r(T_i) = r(U_j)$ (and hence, $d(T_i) = d(U_j)$), $T_i$ has priority over a third subtask $V_k$ if and only if $U_j$ does. All known Pfair scheduling algorithms are weight-consistent.

**Theorem 4.3.** *No weight-consistent Pfair scheduler can guarantee all deadlines on multiprocessors under* (J1) *and* (L1).

*Proof.* Consider a class of task systems consisting of two sets $X$ and $Y$ of tasks of weights $w_1 = 2/5$ and $w_2 = 3/8$, respectively. Whenever subtasks of tasks in $X$ and $Y$ are released simultaneously, a weight-consistent scheduler will favor all subtasks of tasks in $X$ over those in $Y$ or vice versa. We say that $w_1$ *is favored* (analogously for $w_2$) if subtasks of tasks in $X$ are favored. We construct a task system depending on the task weight favored by the scheduler.

**Case 1: $w_1$ is favored.** Consider a dynamic task system consisting of the following types of tasks to be scheduled on 15 processors. (In each of our counterexamples, no subtask is eligible before its PF-window.)

**Type A:** 8 tasks of weight $w_2$ that join at time 0.
**Type B:** 30 tasks of weight $w_1$ that join at time 0 and leave at time 3; each releases one subtask.
**Type C:** 30 tasks of weight $w_1$ that join at time 3.

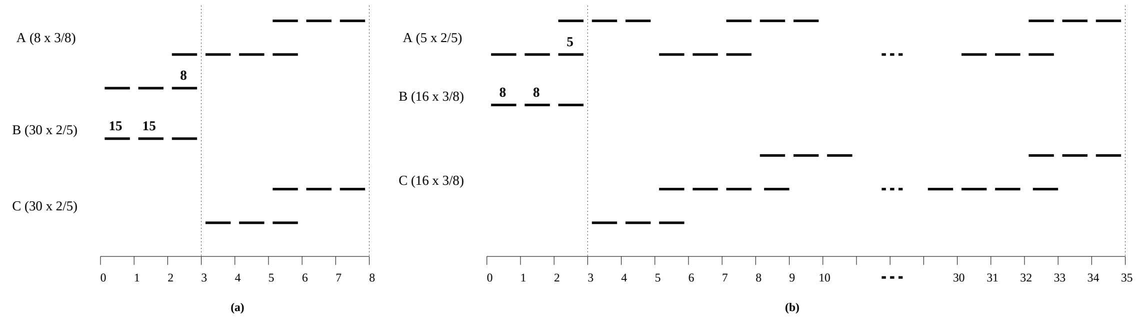



Figure 4.6: Counterexamples demonstrating insufficiency of (L1) for any weight-consistent scheduling algorithm. The notation used in this figure is similar to the one used in Figures 3.3–3.7. The vertical lines depict intervals with excess demand. **(a)** Theorem 4.3. Case 1: Tasks of weight 2/5 are favored at times 0 and 1. **(b)** Theorem 4.3. Case 2: Tasks of weight 3/8 are favored at times 0 and 1.

Because $30w_1 + 8w_2 = 15$, (J1) is satisfied for the type-C tasks. Because $d(T_1) = \lceil \frac{5}{2} \rceil = 3$ for every type-B task $T$, (L1) is also satisfied.

Since $w_1$ is favored, type-B tasks are favored over type-A tasks at times 0 and 1. Hence, the schedule for $[0, 3)$ will be as shown in Figure 4.6(a). Consider the interval $[3, 8)$. Each type-A task has two subtasks remaining for execution, which implies that the type-A tasks need 16 quanta. Similarly, each type-C task also has two subtasks, which implies that the type-C tasks need 60 quanta. However, the total number of quanta in $[3, 8)$ is $15 \cdot (8 - 3) = 75$. Thus, one subtask will miss its deadline at or before time 8.

**Case 2: $w_2$ is favored.** Consider a dynamic task system consisting of the following types of tasks to be scheduled on 8 processors.

**Type A:** 5 tasks of weight $w_1$ that join at time 0.
**Type B:** 16 tasks of weight $w_2$ that join at time 0 and leave at time 3; each releases one subtask.
**Type C:** 16 tasks of weight $w_2$ that join at time 3.

Because $5w_1 + 16w_2 = 8$, (J1) is satisfied for the type-C tasks. Because $d(T_1) = \lceil \frac{8}{3} \rceil = 3$ for every type-B task $T$, (L1) is also satisfied.

Since $w_2$ is favored, type-B tasks are favored over type-A tasks at times 0 and 1. Hence, the schedule for $[0, 3)$ will be as shown in Figure 4.6(b). Consider the interval $[3, 35)$. The number of subtasks of each type-A task that need to be scheduled in $[3, 35)$ is $1 + (35 - 5) \cdot 2/5 = 13$. Similarly, the number of subtasks of each type-C task is

$(35-3)\cdot 3/8 = 12$. The total is $5\cdot 13+16\cdot 12 = 257$, whereas the number of quanta in $[3,35)$ is $(35-3)\cdot 8 = 256$. Thus, one subtask will miss its deadline at or before time 35. ■

The situations described in Theorem 4.3 can be "circumvented" if it can be known at the time a subtask is released whether it is the final subtask of its task. For example, in Figure 4.6(a), if we knew that the first subtask $T_1$ of each type-B task is its last, then we could have given $T_1$ an *effective* $b$-bit of zero. Hence, PD$^2$ would have scheduled it with a lower priority than any type-A task. However, in general, such knowledge may not be available to the scheduler.

The examples in Figure 4.6 show that allowing a light task $T$ to leave at $d(T_i)$ when $b(T_i) = 1$ can lead to deadline misses. We now derive a similar, but stronger, condition for heavy tasks.

**Theorem 4.4.** *If a heavy task $T$ is allowed to leave before $D(T_i)$, where $T_i$ is the last-scheduled subtask of $T$, then there exist task systems that miss a deadline under* PF, PD, *and* PD$^2$.

*Proof.* Consider the following dynamic task system to be scheduled on 35 processors, where $2 \le t \le 4$.

> **Type A:** 9 tasks of weight 7/9 that join at time 0.
> **Type B:** 35 tasks of weight 4/5 that join at time 0 and leave at time $t$; each releases one subtask.
> **Type C:** 35 tasks of weight 4/5 that join at time $t$.

All type-A and type-B tasks have the same PD$^2$ priority at time 0, because each has a deadline at time 2, a $b$-bit of 1, and a group deadline at time 5. Hence, the type-$B$ tasks may be assigned higher priority. Assuming this, Figure 4.7 depicts the schedule for the case of $t = 3$. This counterexample applies to PF and PD as well, because both favor the type-$B$ tasks at time 0.

Consider the interval $[t, t+5)$. Each type-A and type-C task has four subtasks with deadlines in $[t, t+5)$ (see Figure 4.7). Thus, $9\cdot 4 + 35\cdot 4 = 35\cdot 5 + 1$ subtasks must be executed in $[t, t+5)$. Since $35\cdot 5$ quanta are available in $[t, t+5)$, one subtask will miss its deadline. ■

Theorems 4.3 and 4.4 provide us with the following new conditions, which are sufficient when used with PD$^2$ (proved in Section 4.3). Theorems 4.3 and 4.4 also demonstrate the minimality of these conditions.

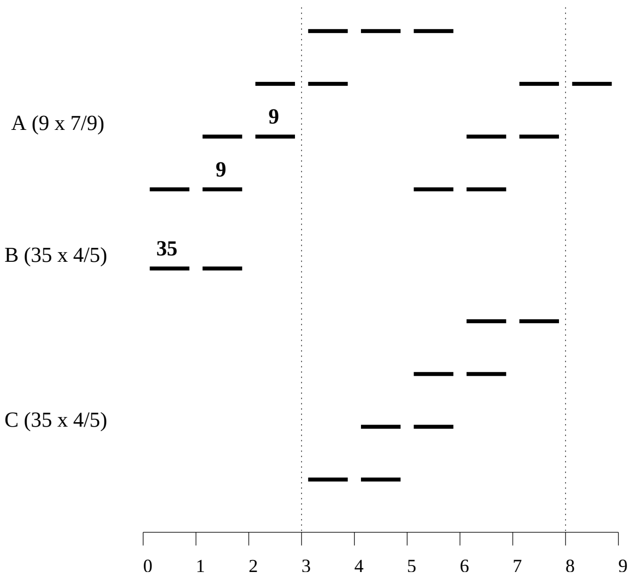



Figure 4.7: Counterexample demonstrating insufficiency of (L1) and tightness of (L2). Theorem 4.4. Each task of weight $4/5$ releases exactly one subtask. All such tasks are allowed to leave at time 3 and re-join immediately. Demand over $[3, 8)$ is more than the available processor time.

**(J2)** *Join condition*: Same as (J1).

**(L2)** *Leave condition*: Let $T_i$ denote the last-scheduled subtask of $T$. If $T$ is light, then $T$ can leave at time $t$ if and only if either $t = d(T_i) \wedge b(T_i) = 0$ or $t > d(T_i)$ holds. If $T$ is heavy, then $T$ can leave at time $t$ if and only if $t \geq D(T_i)$.

(L2) reduces to (L1) if the leaving task is actually periodic or sporadic *and* its final job executes for its worst-case execution requirement. In other words, such a task can leave at the deadline of its last job. To see why, note that if $T_i$ is the last subtask of the final job, then, because consecutive task periods do not overlap, $b(T_i) = 0$ (and hence $D(T_i) = d(T_i)$, if $T$ is heavy). Thus, by (L2), $T$ can leave at time $d(T_i)$, as in (L1).

Any feasible *static* IS task system satisfies (J2) and (L2) by default because no task leaves such a system and (J2) is identical to the feasibility condition. In the next section, we prove that $\mathrm{PD}^2$ correctly schedules any dynamic IS task system satisfying (J2) and (L2). As a corollary, it follows that $\mathrm{PD}^2$ correctly schedules any feasible static IS task system, *i.e.*, it is optimal for scheduling static IS task systems on multiprocessors.

## 4.3 Sufficiency of (J2) and (L2)

In our proof, we consider task systems obtained by removing subtasks from an IS task system. Note that such a task system may no longer be an IS task system (see Figure 4.8). To circumvent this problem, we define a more general model called the *generalized* IS (GIS) task model, and show that PD$^2$ can optimally schedule task systems that belong to this model. In a GIS task system, a task $T$, after releasing subtask $T_i$, may release subtask $T_k$, where $k > i+1$, instead of $T_{i+1}$, with the following restriction: $r(T_k) - r(T_i)$ is at least $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$. In other words, $r(T_k)$ (and hence, $d(T_k)$) is not smaller than what it would have been if $T_{i+1}, T_{i+2}, \ldots, T_{k-1}$ were present and released as early as possible. For the special case where $T_k$ is the first subtask released by $T$, $r(T_k)$ must be at least $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor$.

Thus, the GIS model generalizes the IS model by allowing subtasks to be absent. It follows that for every GIS task system $\tau$, there exists an IS task system $\tau'$ such that $\tau$ can be obtained by simply removing certain subtasks in $\tau'$. Hence, if there exists a schedule for $\tau'$ in which no deadline is missed, then that schedule can be easily modified (by removing subtasks) to obtain a schedule for $\tau$. Therefore, Expression (4.5) is a feasibility condition for GIS task systems as well.

Note that subtask indices for a GIS task are assigned to reflect the missing subtasks. For example, task $T$ in Figure 4.8 releases subtask $T_4$ after releasing $T_2$; $T_3$ is missing and $\theta(T_4) = 0$. Hence, the formulae for subtask release times and deadlines of a GIS task are as in (4.1) and (4.2). Further, the formulae for the $b$-bit and group deadlines are also as defined in (2.11) and (4.7). This implies that the PD$^2$ priority definition of a subtask of a GIS task is the same as for the corresponding IS task.

**Terminology.** An *instance* of a task system is obtained by specifying a unique assignment of release times and eligibility times for each subtask, subject to (4.3). Note that the deadline of a subtask is automatically determined once its release time is fixed (refer to (4.1) and (4.2)). If a task $T$, after executing subtask $T_i$, releases subtask $T_k$, then $T_k$ is called the *successor* of $T_i$ and $T_i$ is called the *predecessor* of $T_k$ (*e.g.*, $T_4$ is $T_2$'s successor in Figure 4.8). The following property is used in our proofs.

**Claim 4.1.** *If subtask $T_k$ is the successor of subtask $T_i$, then $r(T_k) \geq d(T_i) - 1$.*

*Proof.* Note that $\left\lceil \frac{i}{wt(T)} \right\rceil \leq \left\lfloor \frac{i}{wt(T)} \right\rfloor + 1$. Because $k \geq i+1$, $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor \geq \left\lfloor \frac{i}{wt(T)} \right\rfloor$.

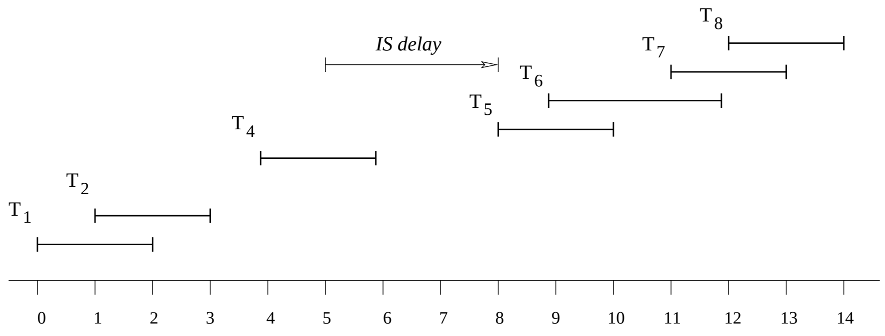



Figure 4.8: The PF-windows of the first eight subtasks of a GIS task $T$ with weight 8/11. Subtask $T_3$ is missing and $T_5$ is released three units late. (Because $T_3$ is missing, this is not an IS task.)

Therefore, $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor \geq \left\lceil \frac{i}{wt(T)} \right\rceil - 1$. By (4.3), $\theta(T_k) \geq \theta(T_i)$. Therefore, $\theta(T_k) + \left\lfloor \frac{k-1}{wt(T)} \right\rfloor \geq \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil - 1$. By (4.1) and (4.2), this implies that $r(T_k) \geq d(T_i) - 1$. ■

### 4.3.1 Displacements

By definition, the removal of a subtask from one instance of a GIS task system results in another valid instance. Let $X^{(i)}$ denote a subtask of any task in a GIS task system $\tau$. Let $S$ denote a schedule of $\tau$ obtained by any scheduling algorithm that schedules on an earliest-pseudo-deadline-first (EPDF) basis (including PF, PD and PD$^2$). Assume that removing $X^{(1)}$ scheduled at slot $t_1$ in $S$ causes $X^{(2)}$ to shift from slot $t_2$ to $t_1$, where $t_1 \neq t_2$, which in turn may cause other shifts. We call this shift a *displacement* and represent it by a four-tuple $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$. A displacement $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ is *valid* if and only if $e(X^{(2)}) \leq t_1$. Because there can be a cascade of shifts, we may have a *chain* of displacements, as illustrated in Figure 4.9.

Removing a subtask may also lead to slots in which some processors are idle. As in Section 3.4, if $k$ processors are idle in slot $t$, then we say that there are $k$ *holes* in slot $t$. Note that holes may exist because of late subtask releases, even if total utilization is $M$.

The lemmas below concern displacements and holes. Lemma 4.4 states that a subtask removal can only cause left-shifts, as in Figure 4.9(b). Lemma 4.5 indicates when a left-shift into a slot with a hole can occur. Lemma 4.6 shows that shifts across a hole cannot occur. Here, $\tau$ is an instance of a GIS task system and $S$ denotes a

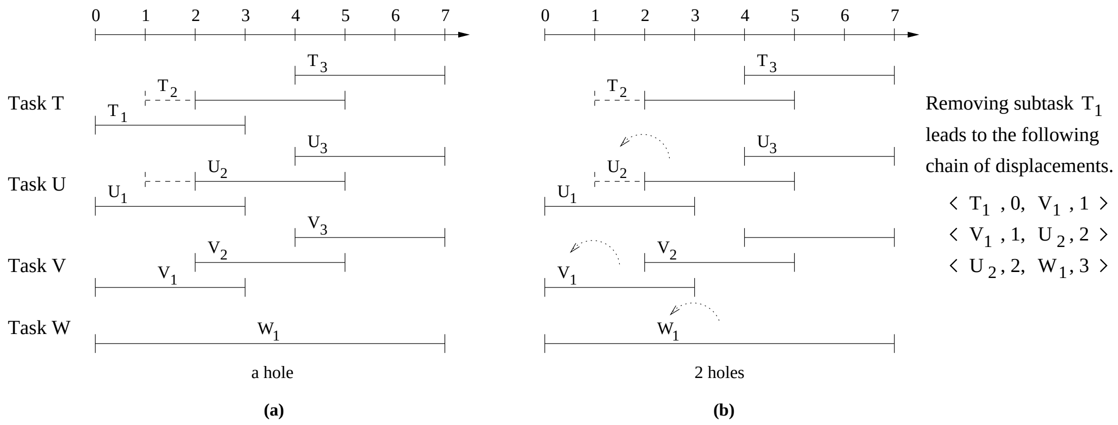



Figure 4.9: A schedule for three tasks of weight 3/7 and one task of weight 1/7 on two processors. The solid lines depict PF-windows; the dashed lines depict IS-windows. Here, only subtasks $T_2$ and $U_2$ are eligible before their PF-windows. Inset **(b)** illustrates the displacements caused by the removal of subtask $T_1$ from the schedule shown in inset **(a)**.

schedule for $\tau$ obtained by a greedy EPDF-based scheduling algorithm. Throughout this section, we assume that ties among subtasks are resolved consistently, *i.e.*, if $\tau'$ is obtained from $\tau$ by a subtask removal, then the relative priorities of two subtasks in $\tau'$ are the same as in $\tau$.

**Lemma 4.4.** *Let $X^{(1)}$ be a subtask that is removed from $\tau$, and let the resulting chain of displacements in $S$ be $C = \Delta_1, \Delta_2, \ldots, \Delta_k$, where $\Delta_i = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \rangle$. Then $t_{i+1} > t_i$ for all $i \in \{1, \ldots, k\}$.*

*Proof.* Let $\tau'$ be the task system instance obtained by removing $X^{(1)}$ from $\tau$, and let $S'$ be its PD$^2$ schedule. Note that the last displacement creates a hole at $t_{k+1}$ in $S'$. Suppose $t_{i+1} > t_i$ is not true for some $i \in \{1, \ldots, k\}$. Let

$$t_j = \mathbf{min}\{t_i \mid t_{i+1} < t_i \ \wedge \ 1 \le i \le k\}.$$

(Informally, the leftmost right-shift occurs when $X^{(j+1)}$ scheduled at $t_{j+1}$ shifts to $t_j$.) We consider two cases depending on whether $j$ is equal to $k$. If $j = k$, then the last displacement will be as shown in Figure 4.10(a). Note that $X^{(k+1)}$ is eligible to be scheduled in slot $t_{k+1}$ in $S'$, because it is scheduled there in $S$ and no subtask (in particular, its predecessor) scheduled before $t_{k+1}$ is shifted to $t_{k+1}$ (by choice of $j$). Because there will be a hole in slot $t_{k+1}$ in $S'$ and $t_{k+1} < t_k$, this contradicts the greedy behavior of the scheduling algorithm.

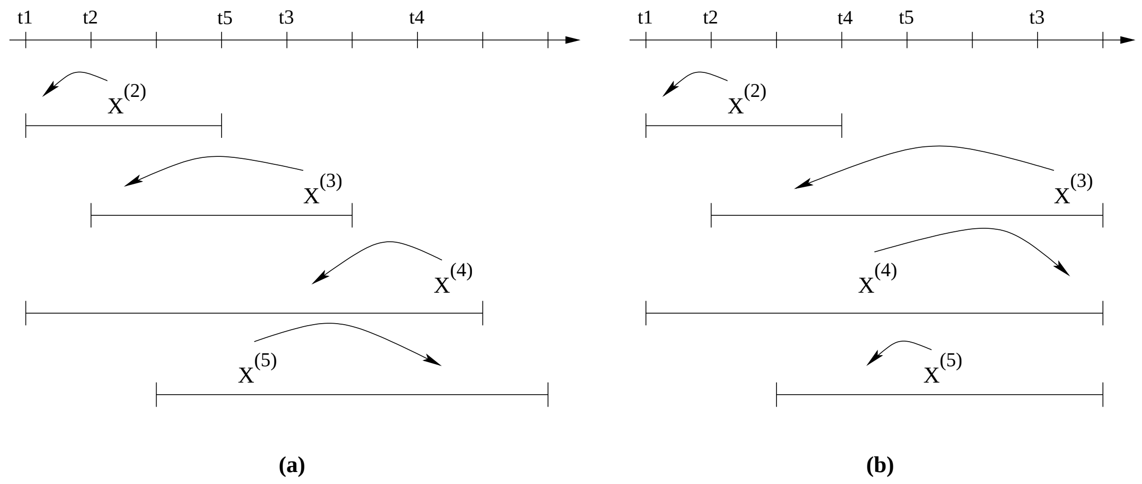



Figure 4.10: Lemma 4.4. A chain of $k = 4$ displacements is shown. **(a)** The leftmost right shift occurs when $X^{(5)}$ shifts from $t_5$ to $t_4$, *i.e.*, $j = k$. **(b)** The leftmost right shift occurs when $X^{(4)}$ shifts from $t_4$ to $t_3$, *i.e.*, $j < k$ (here, $t_j = t_3, t_{j+1} = t_4$, and $t_{j+2} = t_5$).

If $j < k$, then by our choice of $j$, $t_{j+1} < t_j$ and the displacements are as in Figure 4.10(b). By the minimality of $t_j$, $t_{j+2} > t_{j+1}$. Thus, at $t_{j+1}$, $X^{(j+1)}$ was chosen over $X^{(j+2)}$ in $S$. After the displacements, $X^{(j+1)}$ is scheduled at $t_j$ and $X^{(j+2)}$ at $t_{j+1} (< t_j)$. This contradicts our assumption that ties are broken consistently in $S$ and $S'$. Hence, $t_{i+1} > t_i$ for all $i \in \{1, \ldots, k\}$. ■

**Lemma 4.5.** *Let $\Delta = \langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ be a valid displacement in $S$. If $t_1 < t_2$ and there is a hole in slot $t_1$ in $S$, then $X^{(2)}$ is successor of $X^{(1)}$.*

*Proof.* Because $\Delta$ is valid, $e(X^{(2)}) \leq t_1$. Since there is a hole in slot $t_1$ and $X^{(2)}$ is not scheduled there in $S$, $X^{(2)}$ must be the successor of $X^{(1)}$. ■

**Lemma 4.6.** *Let $\Delta = \langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ be a valid displacement in $S$. If $t_1 < t_2$ and there is a hole in slot $t'$ such that $t_1 \leq t' < t_2$ in that schedule, then $t' = t_1$ and $X^{(2)}$ is the successor of $X^{(1)}$.*

*Proof.* Because $\Delta$ is valid, $e(X^{(2)}) \leq t_1$. If $t_1 < t'$, then $e(X^{(2)}) < t'$, implying that $X^{(2)}$ is not scheduled in slot $t_2 > t'$, as assumed, since there is a hole in $t'$. Thus, $t_1 = t'$; by Lemma 4.5, $X^{(2)}$ is the successor of $X^{(1)}$. ■

### 4.3.2 Flows and Lags in GIS Task Systems

The lag of $T$ at time $t$ is defined in the same way as it is defined for periodic tasks. Let $ideal(T,t)$ denote the share that $T$ receives in a fluid schedule in $[0,t)$. Then,

$$lag(T,t) = ideal(T,t) - \sum_{u=0}^{t-1} S(T,u). \tag{4.8}$$

Before defining $ideal(T,t)$, we define $flow(T,u)$, which is the share assigned to task $T$ in slot $u$. $flow(T,u)$ is defined in terms of a function $f(T_i,u)$ (defined in (4.6) that indicates the share assigned to each subtask in each slot. Figure 4.11 shows the values of $f$ for different subtasks of a task of weight 5/16. $flow(T,u)$ is simply defined as $flow(T,u) = \sum_i f(T_i,u)$. Observe that the value of $f(r(T_i),k)$ is the same for a subtask irrespective of whether the task is a periodic task or an IS task. Also, note that $flow(T,u)$ usually equals $wt(T)$, but in certain slots, it may be less than $wt(T)$, so that each subtask of $T$ has a unit share.

The following properties about flows are used in our proof. (We only prove (F1) here to give a flavor of the proof technique used. The other properties are proved in Appendix A.)

**(F1)** For all time slots $t$, $flow(T,t) \le wt(T)$.

*Proof.* (This was proved earlier as part of the proof of Lemma 4.2. However, for the sake of completeness, we repeat the proof here.) We first show that $f(T_i,t) \le wt(T)$. This follows directly from (4.6) if $t \notin \{r(T_i), d(T_i)-1\}$. If $t = r(T_i)$, then

$$\begin{aligned} f(T_i) &= \left(\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1\right) \times wt(T) - (i-1) &&, \text{ by (4.6)} \\ &\le \left(\frac{i-1}{wt(T)} + 1\right) \times wt(T) - (i-1) &&, \lfloor x \rfloor \le x \\ &= wt(T) &&, \text{ by simplification} \end{aligned}$$

If $t = d(T_i) - 1$, then

$$\begin{aligned} f(T_i) &= i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T) &&, \text{ by (4.6)} \\ &\le i - \left(\frac{i}{wt(T)} - 1\right) \times wt(T) &&, \lceil x \rceil \ge x \end{aligned}$$

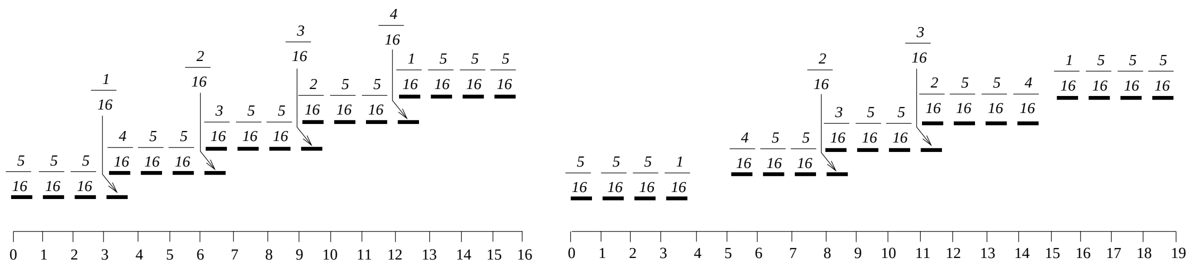



Figure 4.11: Fluid schedule for the first five subtasks ($T_1, \ldots, T_5$) of a task $T$ of weight 5/16. The share of each subtask in each slot of its PF-window is shown. In **(a)**, no subtask is released late; in **(b)**, $T_2$ and $T_5$ are released late. Note that $share(T, 3)$ is either 5/16 or 1/16 depending on when subtask $T_2$ is released.

$$= \quad wt(T) \qquad\qquad , \text{ by simplification}$$

We now only need to consider the time slot in which two consecutive PF-windows overlap. That will be the case when $d(T_i) - 1 = r(T_{i+1})$ for some $i$. In this case, the total flow will be $f(T_i, d(T_i) - 1) + f(T_{i+1}, r(T_{i+1}))$. Thus, $flow(T, d(T_i) - 1)$ is $i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T) + \left(\left\lfloor \frac{i}{wt(T)} \right\rfloor + 1\right) \times wt(T) - i$, which simplifies to $\left(\left\lfloor \frac{i}{wt(T)} \right\rfloor - \left\lceil \frac{i}{wt(T)} \right\rceil + 2\right) \times wt(T)$. Since $d(T_i) - 1 = r(T_{i+1})$, by (4.1) and (4.2), it follows that $\theta(T_i) = \theta(T_{i+1})$ and $\left\lceil \frac{i}{wt(T)} \right\rceil - 1 = \left\lfloor \frac{i}{wt(T)} \right\rfloor$. Therefore, $\left\lfloor \frac{i}{wt(T)} \right\rfloor - \left\lceil \frac{i}{wt(T)} \right\rceil = -1$. Hence, $flow(T, d(T_i) - 1) = wt(T)$. Thus, in all cases, we have $flow(T, t) \leq wt(T)$. ■

**(F2)** Let $T_i$ be a subtask of a GIS task and let $T_k$ be its successor. If $b(T_i) = 1$ and $r(T_k) \geq d(T_i)$, then $flow(T, d(T_i) - 1) + flow(T, d(T_i)) \leq wt(T)$. (For example, in Figure 4.11(b), $flow(T, 3) + flow(T, 4) = 1/16 < 5/16$ and $flow(T, 14) + flow(T, 15) = 5/16$.)

**(F3)** Let $T_i$ be a subtask of a heavy GIS task $T$ such that $b(T_i) = 1$ and let $T_k$ be the successor of $T_i$. If $u \in \{d(T_i), \ldots, D(T_i) - 1\}$ and $u \leq r(T_k)$, then $flow(T, d(T_i)) + flow(T, u) \leq wt(T)$. (This is an extension of (F2) to heavy tasks.)

Since $flow(T, u)$ gives the share of task $T$ in slot $u$, the total share of $T$ over any interval $[0, t)$ $ideal(T, t)$ is defined simply as $\sum_{u=0}^{t-1} flow(T, u)$. Hence, from (4.8), $lag(T, t + 1) = \sum_{u=0}^{t} (flow(T, u) - S(T, u)) = lag(T, t) + flow(T, t) - S(T, t)$. Similarly, the total lag for a schedule $S$ and task system $\tau$ at time $t+1$, denoted by $LAG(\tau, t+1)$, is

$$LAG(\tau, t+1) = LAG(\tau, t) + \sum_{T \in \tau} (\mathit{flow}(T, t) - S(T, t)). \tag{4.9}$$

($LAG(\tau, 0)$ is defined to be 0.) The lemma below is used in our proof.

**Lemma 4.7.** *If $LAG(\tau, t) < LAG(\tau, t+1)$, then there is a hole in slot $t$.*

*Proof.* Let $k$ be the number of subtasks scheduled in slot $t$. Then, by (4.9), $LAG(\tau, t+1) = LAG(\tau, t) + \sum_{T \in \tau} \mathit{flow}(T, t) - k$. If $LAG(\tau, t) < LAG(\tau, t+1)$, then $k < \sum_{T \in \tau} \mathit{flow}(T, t)$. Because $\mathit{flow}(T, t) \leq wt(T)$ (by (P1)), we have $\sum_{T \in \tau} \mathit{flow}(T, t) \leq \sum_{T \in \tau} wt(T)$, which by (4.5) implies that $\sum_{T \in \tau} \mathit{flow}(T, t) \leq M$. Therefore, $k < M$, *i.e.*, there is a hole in slot $t$. ■

### 4.3.3 Proof

We now prove that PD$^2$ correctly schedules any dynamic GIS task system that satisfies (J2) and (L2). We use proof by contradiction, *i.e.*, we show that an assumption contrary to the above leads to a contradiction. Thus, we start by making the following assumption: PD$^2$ misses a deadline for some task system that satisfies (J2) and (L2). Then there exists a time $t_d$ and an instance of a task system $\tau$ as given in Definitions 4.2 and 4.3 below.

**Definition 4.2.** *$t_d$ is the earliest time at which any task system instance misses a deadline under* PD$^2$. ■

**Definition 4.3.** *$\tau$ is an instance of a task system with the following properties.*

**(T1)** *$\tau$ misses a deadline under* PD$^2$ *at $t_d$.*

**(T2)** *No instance of any task system that satisfies* (T1) *releases fewer subtasks in* $[0, t_d)$ *than $\tau$.*

**(T3)** *No instance task system that satisfies both* (T1) *and* (T2) *has a larger rank than $\tau$, where the* rank *of an instance is the sum of the eligibility times of all subtasks with deadlines at most $t_d$.* ■

Existence of $\tau$ follows from the fact that (T1)–(T3) are applied *in sequence*; *e.g.*, $\tau$ is not claimed to be of maximal rank — rather, its rank is maximal *among those task system instances satisfying* (T1) *and* (T2).

By (T1), (T2), and Definition 4.2, exactly one subtask in $\tau$ misses its deadline: if several subtasks miss their deadlines, all but one can be removed and the remaining subtask will still miss its deadline, contradicting (T2).

In the rest of this proof, we use $S$ to denote the PD$^2$ schedule of $\tau$. We now prove several properties about $\tau$ and $S$.

**Lemma 4.8.** *The following properties hold for $\tau$ and $S$.*

**(a)** *For all subtasks $T_i$ in $\tau$, $e(T_i) \geq \mathbf{min}(r(T_i), t)$, where $t$ is the time at which $T_i$ is scheduled in $S$.*

**(b)** *Let $t$ be the time at which $T_i$ is scheduled and let $T_k$ be $T_i$'s successor. If either $d(T_i) > t+1$ or $d(T_i) = t+1 \ \wedge \ b(T_i) = 0$, then $T_k$ is not eligible before $t+1$.*

**(c)** *For all $T_i$, $d(T_i) \leq t_d$.*

**(d)** *There are no holes in slot $t_d - 1$.*

**(e)** *$LAG(\tau, t_d) = 1$.*

**(f)** *$LAG(\tau, t_d - 1) \geq 1$.*

*Proof.* Below, we prove each property separately.

**Proof of (a):** Suppose that $e(T_i) < \mathbf{min}(r(T_i), t)$. Consider the task system instance $\tau'$ obtained from $\tau$ by changing $e(T_i)$ to $\mathbf{min}(r(T_i), t)$. Note that $e(T_i)$ is still at most $r(T_i)$ and $\tau'$'s rank is larger than $\tau$'s. $\tau'$ is feasible because the feasibility proof produces a schedule in which each subtask is scheduled in its PF-window (refer to Section 4.1).

Since PD$^2$ priorities do not depend on the eligibility times, it is easy to see that the relative priorities of the subtasks do not change for any slot $u \in \{0, \ldots, t_d - 1\}$. Hence, $\tau'$ and $\tau$ have identical PD$^2$ schedules. Thus, $\tau'$ misses a deadline at $t_d$, contradicting (T3).

**Proof of (b):** By (4.1) – (4.3), $r(T_k) \geq d(T_i) - 1$. Therefore, if $d(T_i) > t+1$ or $r(T_k) \geq d(T_{i+1})$, then $r(T_k) \geq t+1$. Further, since $T_i$ is scheduled in $[t, t+1)$, $T_k$ is scheduled at or after time $t+1$. Therefore, by part (a), $e(T_k) \geq t+1$

We now consider the case when $d(T_i) = t+1$. Since $b(T_i) = 0$, by (2.11), it follows that $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor = \left\lceil \frac{i}{wt(T)} \right\rceil$. Therefore, by (4.1) – (4.3), $r(T_{i+1}) \geq d(T_i)$. Therefore,

$r(T_j) \geq t+1$, for all $j > i$. In particular, $r(T_k) \geq t+1$, where $T_k$ is $T_i$'s successor. As before, by part (a), it follows that $e(T_k) \geq t+1$.

**Proof of (c):** Suppose $\tau$ contains a subtask $U_j$ with a deadline greater than $t_d$. Since $S$ is obtained using an EPDF-based scheduling algorithm, $U_j$ can be removed without affecting the scheduling of higher-priority subtasks with earlier deadlines. Thus, a deadline will still be missed at $t_d$ after $U_j$'s removal. This contradicts (T2).

**Proof of (d):** Let $U_j$ be the subtask that misses its deadline at $t_d$ in $S$. (Recall that there is only one such subtask.) Because $d(U_j) = t_d$), $d(U_k) \leq t_d - 1$, where $U_k$ is $U_j$'s predecessor. By the minimality of $t_d$, $U_k$ meets its deadline and hence is scheduled before $t_d - 1$. Thus, if there were a hole in slot $t_d - 1$, then $U_j$ would have been scheduled there, in which case it would meet its deadline. Contradiction.

**Proof of (e):** By (4.9), we have

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

The first term on the right-hand side of the above equation is the total share in $[0, t_d)$, which equals the total number of subtasks in $\tau$. The second term equals the number of subtasks scheduled in $S$ over the interval $[0, t_d)$. Since exactly one subtask misses its deadline in $S$, the difference between these two terms is 1, *i.e.*, $LAG(\tau, t_d) = 1$.

**Proof of (f):** By (d), there are no holes in slot $t_d - 1$. Hence, by Lemma 4.7, $LAG(\tau, t_d - 1) \geq LAG(\tau, t_d)$. Therefore, by (e), $LAG(\tau, t_d - 1) \geq 1$.

■

Because $LAG(\tau, 0) = 0$, by part (f) of Lemma 4.8, there exists a time $t$ such that

$$0 \leq t < t_d - 1 \ \wedge \ LAG(\tau, t) < 1 \ \wedge \ LAG(\tau, t+1) \geq 1. \tag{4.10}$$

Without loss of generality, let $t$ be the latest such time, *i.e.*, for all $u$ such that $t < u \leq t_d - 1$, $LAG(\tau, u) \geq 1$. We now show that such a $t$ cannot exist, thus contradicting our starting assumption that $t_d$ and $\tau$ exist.

By (4.10), $LAG(\tau, t) < LAG(\tau, t+1)$. Hence, by Lemma 4.7, *there is at least one hole in slot* $t$. Let $A$ denote the set of tasks scheduled in slot $t$.

Let $B$ denote the set of tasks not in $A$ that are "active" at $t$. A task $U$ is *active* at

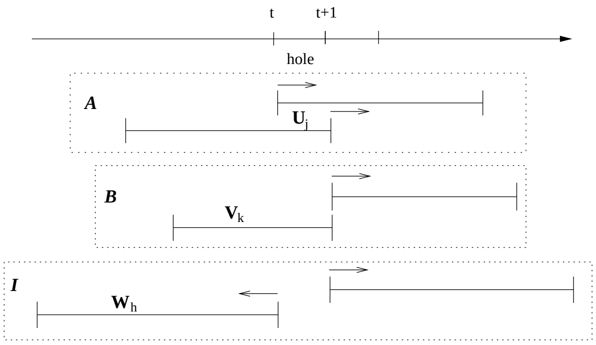



Figure 4.12: Sets $A$, $B$, and $I$. The PF-windows of a sample task of each set are shown. The PF-windows are denoted by line segments. An arrow over a release (respectively, deadline) indicates that the release (respectively, deadline) could be anywhere in the direction of the arrow.

time $t$ if it has a subtask $U_j$ such that $e(U_j) \leq t < d(U_j)$. (A task may be inactive either because it has already left the system or because of a late subtask release.) Consider any task $U \in B$ and let $U_j$ be such that $e(U_j) \leq t < d(U_j)$. Because there is a hole in slot $t$ and no subtask of $U$ is scheduled at time $t$, and because $e(U_j) \leq t < d(U_j)$, $U_j$ must be scheduled before time $t$. Thus, we have the following result.

**Claim 4.2.** *For any subtask $U_j$ of task $U \in B$, if $e(U_j) \leq t < d(U_j)$, then $U_j$ is scheduled before $t$.*

Let $I$ denote the set of the remaining tasks that are not active at time $t$. Figure 4.12 shows how the tasks in $A$, $B$, and $I$ are scheduled.

Of these three sets, set $B$ is the most interesting. As we show below, every task in $B$ must have an IS separation in slot $t$. We use this to prove that $LAG$ reduces below one before time $t_d$. We now prove several properties of set $B$. We first show that in order for $LAG$ to increase during slot $t$, $B$ must be non-empty.

**Lemma 4.9.** *$B$ is non-empty.*

*Proof.* Let the number of the holes in slot $t$ be $h$. Then, $\sum_{T \in \tau} S(T,t) = M - h$. By (4.9), $LAG(\tau, t+1) = LAG(\tau, t) + \sum_{T \in \tau}(\mathit{flow}(T,t) - S(T,t))$. Thus, because $LAG(\tau,t) < LAG(\tau, t+1)$, we have $\sum_{T \in \tau} \mathit{flow}(T,t) > M - h$.

For every $V \in I$, since either $d(V_k) < t$ or $r(V_k) > t$ holds, by (4.6), $\mathit{flow}(V,t) = 0$. It follows that $\sum_{T \in A \cup B} \mathit{flow}(T,t) > M - h$. Therefore, by (F1), $\sum_{T \in A \cup B} wt(T) > M - h$.

Because the number of tasks scheduled in slot $t$ is $M - h$, $|A| = M - h$. Because $wt(T) \leq 1$ for any task $T$, $\sum_{T \in A} wt(T) \leq M - h$. Thus, $\sum_{T \in B} wt(T) > 0$. Hence, $B$ is not empty. ■

In the proof of Lemmas 4.10, 4.11, and 4.13, we use the following technique to prove the required result: if the required condition is not satisfied, then a subtask can be removed without causing the missed deadline at $t_d$ to be met, thus contradicting (T2).

**Lemma 4.10.** *Let $U$ be any task in $B$. Let $U_j$ be the subtask with the largest index such that $e(U_j) \leq t < d(U_j)$. Then, $d(U_j) = t + 1 \ \wedge\ b(U_j) = 1$.*

*Proof.* By Claim 4.2, $U_j$ must be scheduled before $t$. By (4.10), $t < t_d$. Hence, $U_j$ does not miss its deadline. From the lemma statement, we have $d(U_j) \geq t+1$. Suppose that the following holds.

$$d(U_j) > t + 1 \text{ or } d(U_j) = t + 1 \ \wedge\ b(U_j) = 0 \tag{4.11}$$

Under these assumptions, we show that $U_j$ can be removed and a deadline will still be missed at $t_d$, contradicting (T2). Let the chain of displacements caused by removing $U_j$ be $\Delta_1, \Delta_2, \ldots, \Delta_k$, where $\Delta_i = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \rangle$ and $X^{(1)} = U_j$. By Lemma 4.4, $t_{i+1} > t_i$ for $1 \leq i \leq k$.

Note that at slot $t_i$, the priority of subtask $X^{(i)}$ is at least that of $X^{(i+1)}$, because $X^{(i)}$ was chosen over $X^{(i+1)}$ in $S$. Thus, because $X^{(1)} = U_j$, by (4.11), for each subtask $X^{(i)}, 1 \leq i \leq k+1$, either $d(X^{(i)}) > t + 1$ or $d(X^{(i)}) = t + 1 \ \wedge\ b(X^{(i)}) = 0$. Therefore, by part (b) of Lemma 4.8, the following property holds.

**(E)** The eligibility time of the successor of $X^{(i)}$ (if it exists in $\tau$) is at least $t + 1$ for all $i \in \{1, \ldots, k+1\}$.

We now show that the displacements do not extend beyond slot $t$. Assume, to the contrary, that $t_{k+1} > t$. Consider $h \in \{2, \ldots, k+1\}$ such that $t_h > t$ and $t_{h-1} \leq t$, as depicted in Figure 4.13(a). Such an $h$ exists because $t_1 < t < t_{k+1}$. Because there is a hole in slot $t$ and $t_{h-1} \leq t < t_h$, by Lemma 4.6, $t_{h-1} = t$ and $X^{(h)}$ must be $X^{(h-1)}$'s successor. Therefore, by (E), $e(X^{(h)}) \geq t + 1$. This implies that $\Delta_{h-1}$ is not valid.

Thus, the displacements do not extend beyond slot $t$, implying that no subtask scheduled after $t$ is left-shifted. Hence, a deadline is still missed at time $t_d$, contradicting (T2). Hence, $d(U_j) = t + 1 \ \wedge\ b(U_j) = 1$. ■

We now consider two separate cases depending on whether $B$ contains a light task.

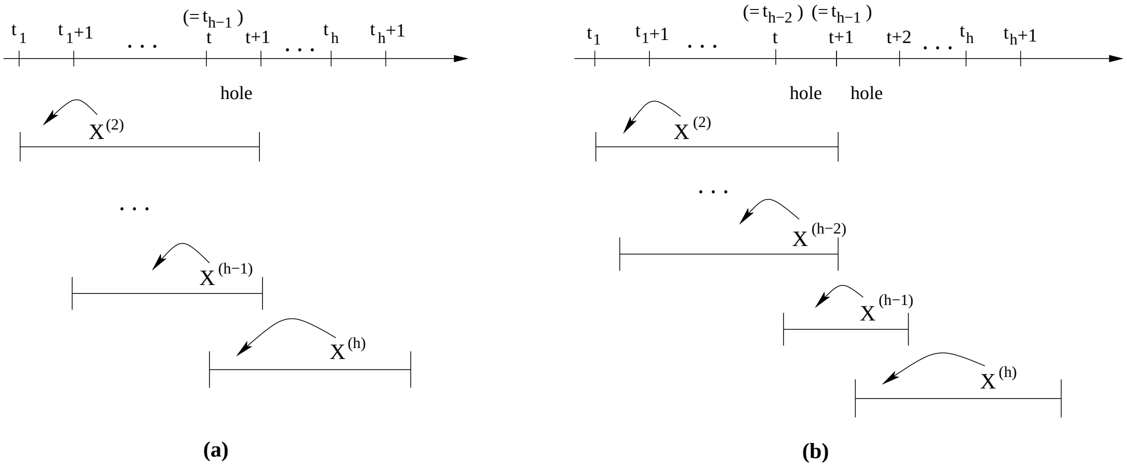



Figure 4.13: IS-windows are denoted by line segments. **(a)** Lemma 4.10. $X^{(h)}$ must be the successor of $X^{(h-1)}$ because there is a hole in slot $t$. **(b)** Lemma 4.11. If there is a hole in both slots $t$ and $t+1$, then $X^{(h-2)}$ and $X^{(h-1)}$ must be scheduled at $t$ and $t+1$ in $S$, respectively. Also, in $\tau$, $X^{(h)}$ must be the successor of $X^{(h-1)}$, which in turn, must be the successor of $X^{(h-2)}$.

### At Least One Task in $B$ is Light

The following property (proved in Appendix A) is used in the proof of Lemma 4.11.

**(L)** For a light task $T$, if $T_k$ is the successor of $T_i$, then $d(T_k) \geq d(T_i) + 2$.

**Lemma 4.11.** *If $B$ has at least one light task, then there is no hole in slot $t+1$.*

*Proof.* By (4.10), $t < t_d - 1$, and therefore, $t + 1 \leq t_d - 1$. Suppose that there is a hole in slot $t+1$. By part (d) of Lemma 4.8, $t+1 < t_d - 1$, *i.e.*,

$$t + 2 \leq t_d - 1. \tag{4.12}$$

Let $U$ be a light task in $B$ and let $U_j$ be the subtask of $U$ with the largest index such that $e(U_j) \leq t < d(U_j)$. Our approach is the same as in the proof of Lemma 4.10. Let the chain of displacements caused by removing $U_j$ be $\Delta_1, \Delta_2, \ldots, \Delta_k$, where $\Delta_i = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1}\rangle$ and $X^{(1)} = U_j$. By Lemma 4.4, we have $t_{i+1} > t_i$ for all $i \in [1, k]$. Also, the priority of $X^{(i)}$ is at least that of $X^{(i+1)}$ at $t_i$, because $X^{(i)}$ was chosen over $X^{(i+1)}$ in $S$. Because $U$ is light and $d(U_j) = t+1 \ \wedge \ b(U_j) = 1$ (by Lemma 4.10), this implies the following.

**(P)** For all $i \in \{1, \ldots, k+1\}$, either **(i)** $d(X^{(i)}) > t+1$ or **(ii)** $d(X^{(i)}) = t+1$ and $X^{(i)}$ is the subtask of a light task.

Suppose the chain of displacements extends beyond $t+1$, *i.e.*, $t_{k+1} > t+1$. Consider $h \in \{1, \ldots, k+1\}$ such that that $t_h > t+1$ and $t_{h-1} \leq t+1$. Because there is a hole in slot $t+1$ and $t_{h-1} \leq t+1 < t_h$, by Lemma 4.6, $t_{h-1} = t+1$ and $X^{(h)}$ is the successor of $X^{(h-1)}$. Similarly, because there is a hole in slot $t$, $t_{h-2} = t$ and $X^{(h-1)}$ is the successor of $X^{(h-2)}$. This is illustrated in Figure 4.13(b).

By (P), either $d(X^{(h-2)}) > t+1$ or $d(X^{(h-2)}) = t+1$ and $X^{(h-2)}$ is the subtask of a light task. In either case, $d(X^{(h-1)}) > t+2$. To see why, note that if $d(X^{(h-2)}) > t+1$, then because $X^{(h-1)}$ is the successor of $X^{(h-2)}$, by (4.2), $d(X^{(h-1)}) > t+2$. On the other hand, if $d(X^{(h-2)}) = t+1$ and $X^{(h-2)}$ is the subtask of a light task, then, by (L), $d(X^{(h-1)}) > t+2$.

Now, because $X^{(h-1)}$ is scheduled at $t+1$, by part (b) of Lemma 4.8, the successor of $X^{(h-1)}$ is not eligible before $t+2$, *i.e.*, $e(X^{(h)}) \geq t+2$. This implies that the displacement $\Delta_{h-1}$ is not valid. Thus, the chain of displacements cannot extend beyond time $t+2$. Hence, because $t+2 \leq t_d - 1$ (by (4.12)), removing $U_j$ cannot cause a missed deadline at $t_d$ to be met. This contradicts (T2). Hence, there is no hole in slot $t+1$. ■

**Lemma 4.12.** *If $B$ has at least one light task, then $LAG(\tau, t+2) < 1$.*

*Proof.* Let the number of holes in slot $t$ be $h$. We now derive some properties about the *flow* values in slots $t$ and $t+1$.

By the definition of $I$, only tasks in $A \cup B$ are active at time $t$. Thus, $\sum_{T \in \tau} \mathit{flow}(T,t) = \sum_{T \in A \cup B} \mathit{flow}(T,t)$. Since $wt(T) \leq 1$ for any $T$, we have $\sum_{T \in A} wt(T) \leq |A|$. Thus, by (F1), $\sum_{T \in A} \mathit{flow}(T,t) \leq |A|$. Now, because there are $h$ holes in slot $t$, $M-h$ tasks are scheduled at $t$, *i.e.*, $|A| = M-h$. Thus, $\sum_{T \in A} \mathit{flow}(T,t) \leq M-h$ and

$$\sum_{T \in \tau} \mathit{flow}(T,t) \leq M - h + \sum_{T \in B} \mathit{flow}(T,t). \tag{4.13}$$

Consider $U \in B$. Let $U_j$ be the subtask of $U$ with the largest index such that $e(U_j) \leq t < d(U_j)$. Let $C$ denote the set of such subtasks for all tasks in $B$. Then, by Lemma 4.10,

$$\text{for all } U_j \in C,\ d(U_j) = t+1 \ \wedge\ b(U_j) = 1. \tag{4.14}$$

If $U$ is heavy, then this would imply that $D(U_j) > t+1$. (By the definition of a group deadline, for any subtask $T_i$ of a heavy task $T$, $D(T_i) = d(T_i)$ holds if and only if $b(T_i) = 0$.) Thus, the leave condition (L2) is not satisfied at time $t+1$, and hence no task in $B$ leaves at time $t+1$.

Let $A'$ denote the tasks in $A$ that are active at time $t+1$. Similarly, let $I'$ denote the tasks in $A'$ that are active at time $t+1$. Then, the set of active tasks at time $t+1$ is $A' \cup I' \cup B$. Thus, by the join condition (J2),

$$\sum_{T \in A' \cup I' \cup B} wt(T) \leq M. \tag{4.15}$$

Also, $\sum_{T \in \tau} \mathit{flow}(T, t+1) = \sum_{T \in A' \cup I' \cup B} \mathit{flow}(T, t+1)$. By (F1), this implies that $\sum_{T \in \tau} \mathit{flow}(T, t+1) \leq \sum_{T \in A' \cup I'} wt(T) + \sum_{T \in B} \mathit{flow}(T, t+1)$. Thus, by (4.13),

$$\sum_{T \in \tau} (\mathit{flow}(T,t) + \mathit{flow}(T,t+1)) \leq M - h + \sum_{T \in A' \cup I'} wt(T) + \sum_{T \in B} (\mathit{flow}(T,t) + \mathit{flow}(T,t+1)) \tag{4.16}$$

Consider $U_j \in C$ (hence, $U \in B$). Let $U_k$ denote the successor of $U_j$. Since $U_j$ is the subtask with the largest index such that $e(U_j) \leq t < d(U_j)$, we have $e(U_k) \geq t+1$. Hence, $r(U_k) \geq t+1$. By (4.14), we have $d(U_j) = t+1$. Therefore, by (F2), $\mathit{flow}(U,t) + \mathit{flow}(U,t+1) \leq wt(U)$ for each $U \in B$. By (4.16), this implies that $\sum_{T \in \tau} (\mathit{flow}(T,t) + \mathit{flow}(T,t+1)) \leq M - h + \sum_{T \in A' \cup I' \cup B} wt(T)$. Thus, from (4.15), it follows that

$$\sum_{T \in \tau} (\mathit{flow}(T,t) + \mathit{flow}(T,t+1)) \leq M - h + M. \tag{4.17}$$

By the statement of the lemma, $B$ contains at least one light task. Therefore, by Lemma 4.11, there is no hole in slot $t+1$. Since there are $h$ holes in slot $t$, we have $\sum_{T \in \tau} (S(T,t) + S(T,t+1)) = M - h + M$.

Hence, by (4.17), $\sum_{T \in \tau} (\mathit{flow}(T,t) + \mathit{flow}(T,t+1)) \leq \sum_{T \in \tau} (S(T,t) + S(T,t+1))$. Using this relation in the identity (obtained from (4.9)), $LAG(\tau, t+2) = LAG(\tau, t) + \sum_{T \in \tau} (\mathit{flow}(T,t) + \mathit{flow}(T,t+1)) - \sum_{T \in \tau} (S(T,t) + S(T,t+1))$, and the fact that $LAG(\tau, t) < 1$, we obtain $LAG(\tau, t+2) < 1$. ■

**All Tasks in $B$ are Heavy.**

We now extend Lemmas 4.11 and 4.12 to the case in which $B$ consists solely of heavy tasks. The following lemma is the counterpart of Lemma 4.11.

**Lemma 4.13.** *Let $U$ be a heavy task in $B$ and let $U_j$ be the subtask of $U$ with the largest index such that $e(U_j) \leq t < d(U_j)$. Then, there exists a slot with no holes in $[d(U_j), \mathbf{min}(D(U_j), t_d))$.*

*Proof.* By Lemma 4.10, $d(U_j) = t + 1 \wedge b(U_j) = 1$. By (4.10), $t < t_d - 1$. Therefore $d(U_j) \leq t_d - 1$. If $\mathbf{min}(D(U_j), t_d) = t_d$, then by part (f) of Lemma 4.8, slot $t_d - 1$ satisfies the stated requirement. In the rest of the proof, assume that $D(U_j) < t_d$. Let $v = D(U_j)$. Since $b(U_j) = 1$, by the definition of $D$, $D(U_j) > d(U_j)$, *i.e.*,

$$t + 1 < v. \tag{4.18}$$

Suppose that the following property holds.

**(H)** There is a hole in slot $u$ for all $u \in \{t, \ldots, v - 1\}$.

Given (H), we show that removing $U_j$ does not cause the missed deadline to be met, contradicting (T2). Let $\Delta_1, \Delta_2, \ldots, \Delta_k$ be the chain of displacements caused by removing $U_j$, where $\Delta_i = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \rangle$, $X^{(1)} = U_j$, and $t_1$ is the slot in which $U_j$ is scheduled. By Lemma 4.4, $t_{i+1} > t_i$ for all $i \in \{1, \ldots, k - 1\}$. Also, the priority of $X^{(i)}$ is at least that of $X^{(i+1)}$ at $t_i$, because $X^{(i)}$ was chosen over $X^{(i+1)}$ at $t_i$ in $S$. Thus, by Lemma 4.10, for all $i \in \{2, \ldots, k + 1\}$, one of the following holds:

**(a)** $d(X^{(i)}) > t + 1$,

**(b)** $d(X^{(i)}) = t + 1 \ \wedge\ b(X^{(i)}) = 0$, or

**(c)** $d(X^{(i)}) = t + 1 \ \wedge\ b(X^{(i)}) = 1 \ \wedge\ D(X^{(i)}) \leq v$.

We now show that the displacements do not extend beyond slot $v - 1$ (which implies that $U_j$ can be removed without causing the missed deadline to be met). Suppose, to the contrary, they do extend beyond slot $v - 1$, *i.e.*, $t_{k+1} > v - 1$.

Let $t_g$ be the largest $t_i$ such that $t_i < t$ and let $t_h$ be the smallest $t_i$ such that $t_i > v - 1$. (Note that such a $t_g$ exists because $t_1 < t$.) Then, by (H), there are holes in all slots in $[t_{g+1}, t_{h-1}]$. Thus, by Lemma 4.6,

$$\forall i \in [g+1, h-1],\ X^{(i+1)} \text{ is the successor of } X^{(i)}. \tag{4.19}$$

Also, $t_{i+1} = t_i + 1$ for all $i \in \{g + 1, \ldots, h - 2\}$.

$$t_{g+1} = t \ \wedge\ t_{h-1} = v - 1 \tag{4.20}$$

$$\forall i \in \{g + 1, \ldots, h - 1\},\ t_i = t + i - (g + 1) \tag{4.21}$$

Earlier, we showed that one of (a) – (c) holds for all $i \in [2, k + 1]$. If either $d(X^{(g+1)}) > t + 1$ or $d(X^{(g+1)}) = t + 1 \ \wedge\ b(X^{(g+1)}) = 0$, then since $X^{(g+1)}$ is scheduled

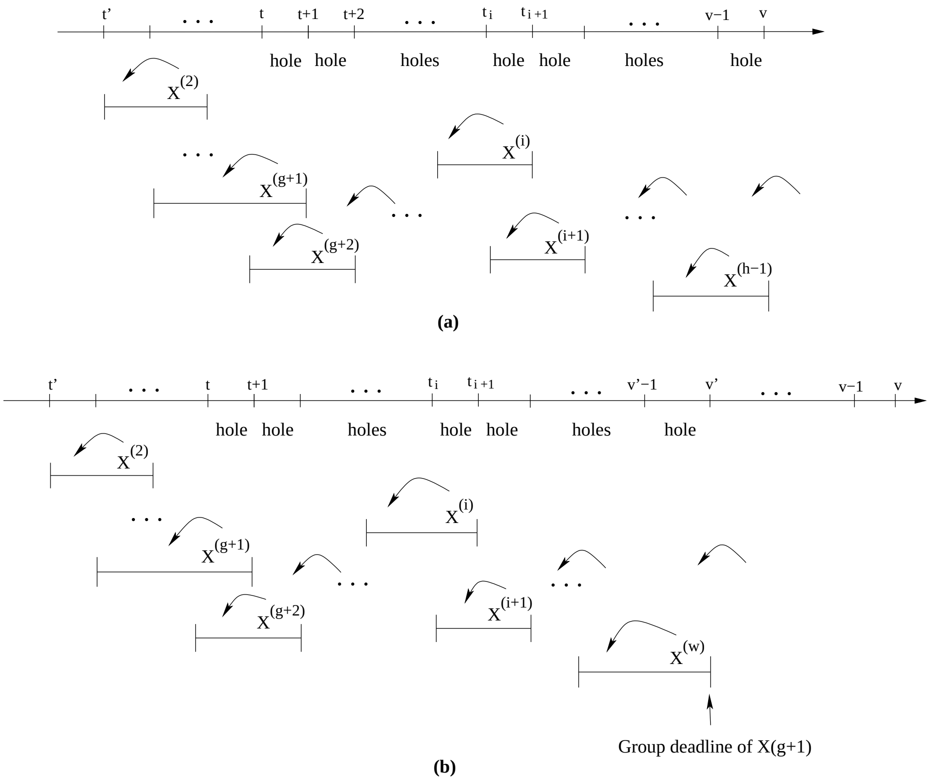



Figure 4.14: Lemma 4.13. **(a)** There are holes in all slots in $[t, v)$. $X^{(i)}$ scheduled at $t_i$ displaces $X^{(i-1)}$ scheduled at $t_{i-1}$. By (4.21), the $t_i$'s are consecutive and satisfy $t_i = t + i - (g + 1)$. Further, $X^{(h-1)}$ is the subtask scheduled in slot $v - 1$. **(b)** Case 2. $D(X^{(g+1)}) = v'$. Hence, either $d(X^{(w)}) = v' \wedge b(X^{(w)}) = 0$ (as depicted) or $d(X^{(w)}) > v'$.

at $t$, by Lemma 4.8, part (b), $e(X^{(g+2)}) \geq t + 1$ (recall that, by (4.19), $X^{(g+2)}$ is the successor of $X^{(g+1)}$). In other words, the displacement $\Delta_g$ is not valid. Therefore,

$$d(X^{(g+1)}) = t + 1 \wedge b(X^{(g+1)}) = 1 \wedge D(X^{(g+1)}) \leq v. \tag{4.22}$$

We now consider two cases. In each, we show that the displacements do not extend beyond $v - 1$, as desired.

**Case 1: $X^{(g+1)}$ is the subtask of a light task.** By (4.18), $t + 1 \leq v - 1$ and hence, by (H), there is a hole in both $t$ and $t + 1$. Also, by (4.20) and (4.21), we have $v - 1 = t + (h - 1) - (g + 1) = t + h - g - 2$. Because $t < v - 1$ (by (4.18)), we have $h > g + 2$, *i.e.*,

$$h \geq g + 3.$$

Because $X^{(g+1)}$ is the subtask of a light task, the reasoning used in the proof of Lemma 4.11 applies. Thus, the displacement $\Delta_{g+2}$ is not valid. Hence, the displacements do not extend beyond slot $t+1$ (and hence, slot $v-1$).

**Case 2: $X^{(g+1)}$ is the subtask of a heavy task.** Let $v' = D(X^{(g+1)})$. By (4.22), $v' \le v$. We now show that the displacements cannot extend beyond slot $v'-1$ (and hence, slot $v-1$). By (4.21), $X^{(i)}$ is scheduled in slot $t+i-(g+1)$ in $S$ for all $i \in \{g+1,\ldots,h-1\}$. By (4.19), all $X^{(i)}$ where $g+1 \le i \le h$ are subtasks of the same heavy task. We now show that the displacement $\Delta_{v'-1-t+(g+1)} (= \Delta_{v'-t+g})$ is not valid. Let $w = v'-t+g$.

By (4.21), $t_w = v'-1$. Because $X^{(i)}$ is scheduled at $t_i$, the subtask scheduled at $v'-1$ is $X^{(w)}$. Since $X^{(i+1)}$ is the successor of $X^{(i)}$, by (4.2), $d(X^{(i)}) > d(X^{(i-1)})$ for all $i \in [g+2, w]$. Because $d(X^{(g+1)}) = t+1$ (by (4.22)),

$$\forall i \in \{g+1,\ldots,w\},\ d(X^{(i)}) \ge t+i-g. \tag{4.23}$$

In particular, $d(X^{(w)}) \ge v'$.

We now show that if $d(X^{(w)}) = v'$, then $b(X^{(w)}) = 0$. In this case, because $d(X^{(w-1)}) < d(X^{(w)})$, we have $d(X^{(w-1)}) < v'$. By (4.23), $d(X^{(w-1)}) \ge v'-1$. Therefore, $d(X^{(w-1)}) = v'-1$. Similarly, by induction, $d(X^{(i)}) = u+i-g)$ for all $i \in \{g+1,\ldots,w\}$. (Refer to Figure 4.14(b).) Because $D(X^{(g+1)}) = v'$, by the definition of $D$, $b(X^{(v'-u+g+1)}) = 0$. (In this case, the group deadline corresponds to the last slot of a window of length two.)

Thus, either $d(X^{(w)}) > v'$ or $d(X^{(w)}) = v' \wedge b(X^{(w)}) = 0$. Since $X^{(w)}$ is scheduled at $v'-1$, by Lemma 4.8, part (b), the eligibility time of the successor of $X^{(w)}$ is at least $v'$. Hence, $\Delta_w$ is not valid. Thus, the displacements do not extend beyond slot $v'-1$. ■

Lemma 4.14 below extends Lemma 4.12 by allowing $B$ to consist solely of heavy tasks. The following results are used in its proof.

**Claim 4.3.** *If $U_j$ is scheduled in slot $u$, where $0 \le u < t_d$ and $u \le d(U_j)$, and if there is a hole in slot $u$, then $d(U_j) = u+1$.*

*Proof.* Because $u < t_d$, by Definition 4.2, no deadline is missed in $[0, u+1)$. Because $U_j$ is scheduled in slot u, *i.e.*, $[u, u+1)$, we have $d(U_j) \ge u+1$. Suppose that $d(U_j) > u+1$. Then, by part (b) of Lemma 4.8, the successor of $U_j$ (if it exists) is not eligible before

$u+1$. Hence, by Lemma 4.5, we can remove $U_j$ and no displacements will result, *i.e.*, a deadline is still missed at $t_d$, contradicting (T2). Therefore, $d(U_j) = u+1$. ■

**Claim 4.4.** *Suppose there is a hole in slot $u \in \{0, \ldots, t_d - 1\}$. Let $U_j$ be a subtask scheduled at $t' \leq u$. If the eligibility time of the successor of $U_j$ is at least $u+1$, then $d(U_j) \leq u+1$.*

*Proof.* If $t' = u$, then by Claim 4.3, $d(U_j) = u+1$. On the other hand, if $t' < u$ and $d(U_j) \geq u$, then by Lemma 4.10, $d(U_j) = u+1$. ■

**Lemma 4.14.** *There exists $v \in \{t+2, \ldots, t_d\}$ such that $LAG(\tau, v) < 1$.*

*Proof.* Because $LAG(\tau, t) < 1$ and $LAG(\tau, t+1) \geq 1$ (by (4.10)),

$$LAG(\tau, t) < LAG(\tau, t+1). \tag{4.24}$$

Thus, by Lemma 4.7, we have the following property.

**(H)** There is at least one hole in slot $t$.

Let $A, B$, and $I$ be as defined in the proof of Lemma 4.12. If any task in $B$ is light, then by Lemma 4.12, $LAG(\tau, t+2) \leq 0$, which establishes our proof obligation. We henceforth assume all tasks in $B$ are heavy.

Let $U$ be any task in $B$. Let $U_j$ be the subtask with the largest index such that $e(U_j) \leq t < d(U_j)$. Let $C$ denote the set of such subtasks of all tasks in $B$. Then, by Lemma 4.10,

$$\forall\ U_j \in C,\ d(U_j) = t+1\ \wedge\ b(U_j) = 1. \tag{4.25}$$

Let $L_i$ be the lowest-priority subtask in $C$. Then,

$$\forall U_j \in C,\ d(U_j) = t+1\ \wedge\ b(U_j) = 1\ \wedge\ D(U_j) \geq D(L_i). \tag{4.26}$$

By Lemma 4.13, there is a slot in $[t, \mathbf{min}(D(L_i), t_d))$ with no hole. Let $u$ be as follows.

**(U)** $u$ is the earliest slot in $[t, \mathbf{min}(D(L_i), t_d))$ with no hole.

Figure 4.15 depicts this situation. By (U) and (H),

$$u \geq t+1, \tag{4.27}$$

and there are holes in all slots in $\{t, \ldots, u-1\}$. We now establish the following property about tasks in $B$.

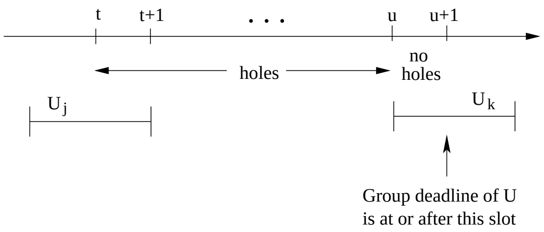



Figure 4.15: Lemma 4.14. $U_j \in C$ and $U_k$ is the successor of $U_j$. There is a hole in each slot in $[t, u)$ and there is no hole in slot $u$. The earliest time at which $U_k$'s PF-window starts is $u$, *i.e.*, $r(U_k) \geq u$.

**Claim 4.5.** *All tasks in $B$ are inactive over the interval $[t+1, u)$.*

*Proof.* If the interval $[t+1, u)$ is empty, then the claim is vacuously true, so assume it is nonempty. Let $V$ be any task in $B$. We first show that no subtask of $V$ is scheduled in $[t, u)$.

Note that because $V \in B$, no subtask of $V$ is scheduled in slot $t$. Let $V_i$ be the earliest subtask of $V$ scheduled in $[t+1, u)$ and let $v$ be the slot in which it is scheduled. Because there is hole in slot $v$, by Claim 4.3, $d(V_i) = v + 1$. By (4.1) and (4.2), this implies that $r(V_i) \leq v$. If $r(V_i) < v$, then $e(V_i) < v$. Thus, because there are holes in all slots in $\{t, \ldots, v-1\}$, it should have been scheduled earlier. Therefore, $r(V_i) = v$, which implies that $wt(V) = 1$. However, this contradicts the fact that some subtask of $V$ has a $b$-bit of 1 (by (4.25)). Hence, no subtask of any task in $B$ is scheduled in $[t, u)$ (see Figure 4.15). Moreover, because there are holes in all slots in $[t, u)$, the earliest slot after $t$ at which a subtask of a task in $B$ is eligible to be scheduled is $u$. By (4.25), this implies that all the tasks in $B$ are inactive in $[t+1, u-1]$. ■

For any $U_j \in C$, by (4.26) and (U), $D(U_j) > u$. Therefore, by the leave condition (L2), task $U$ cannot leave before time $u+1$. Thus, no task in $B$ can leave before time $u+1$.

Let $U_j$ be any subtask in $C$, and let $U_k$ be the successor of $U_j$. By Claim 4.5, $r(U_k) \geq u$. Furthermore, by (4.25) – (4.27) and (U), $d(U_j) = t+1 \leq u < D(U_j)$. Hence, by (F3), $\mathit{flow}(U,t) + \mathit{flow}(U,u) \leq wt(U)$. Because this argument applies to all tasks in $B$, we have

$$\forall U \in B,\ \mathit{flow}(U,t) + \mathit{flow}(U,u) \leq wt(U). \tag{4.28}$$

We now show that $LAG$ is non-increasing over $[t+1, u)$.

**Claim 4.6.** *$LAG(\tau, v+1) \leq LAG(\tau, v)$ for all $v \in \{t+1, \ldots u-1\}$,*

*Proof.* If $\{t+1, \ldots, u\}$ is empty, then the claim is vacuously true, so assume it is nonempty. Suppose for some $v \in \{t+1, \ldots, u-1\}$, $LAG(\tau, v+1) > LAG(\tau, v)$. Then, by Lemma 4.9, there exists a task that is active at $v$ but not scheduled at $v$. Let $V$ be one such task and let $V_k$ be the subtask with the largest index such that

$$e(V_k) \leq v < d(V_k). \tag{4.29}$$

Because no subtask of $V$ is scheduled at $v$ and because there is a hole at $v$, $V_k$ is scheduled before $v$. By (U), there is a hole at $v-1$; moreover, because $t+1 \leq v \leq u-1$, we have $v-1 \in \{t, \ldots, u-2\} \subseteq \{0, \ldots, t_d - 1\}$. Hence, by Claim 4.4, we have $d(V_k) \leq v$, which contradicts (4.29). Therefore, $LAG(\tau, v+1) \leq LAG(\tau, v)$ for all $v \in \{t+1, \ldots, u-1\}$. ■

We now show that $LAG(\tau, u+1) \leq 0$, which establishes our proof obligation.

For each $v \in \{t, \ldots, u\}$, let $H_v$ denote the number of holes in slot $v$. Then, $M - H_v$ tasks are scheduled in slot $v$. Also, let $I_v$ ($A_v$) denote the tasks in $I$ ($A$) that are active at $v$.

By (4.9) and Claim 4.6, $\sum_{T \in \tau} \mathit{flow}(T, v) \leq \sum_{T \in \tau} S(T, v)$. Therefore,

$$\forall\, v \in \{t+1, \ldots, u-1\},\ \sum_{T \in \tau} \mathit{flow}(T, v) \leq M - H_v. \tag{4.30}$$

By the join condition (J2) and by (4.5), we have $\sum_{T \in B \cup I_u \cup A_u} wt(T) \leq M$. Hence, by (4.28) and (F1), we get $\sum_{T \in B} (\mathit{flow}(T, t) + \mathit{flow}(T, u)) + \sum_{T \in I_u \cup A_u} \mathit{flow}(T, u) \leq M$. Thus,

$$\sum_{T \in B} \mathit{flow}(T, t) + \sum_{T \in B \cup I_u \cup A_u} \mathit{flow}(T, u) \leq M. \tag{4.31}$$

Because the tasks in $A (= A_t)$ are the ones scheduled in slot $t$, the number of tasks in set $A_t$ is $M - H_t$. Hence, by (F1) and because the weight of each task is at most one,

$$\sum_{T \in A_t} \mathit{flow}(T, t) \leq \sum_{T \in A_t} wt(T) \leq M - H_t. \tag{4.32}$$

We are now ready to show that $LAG(\tau, u+1) \leq 0$. Because $S(T, v) = M - H_v$, by (4.9), $LAG(\tau, u+1) - LAG(\tau, t) = R$, where $R = \sum_{v=t}^{u} \left( \sum_{T \in \tau} \mathit{flow}(T, v) \right) - \sum_{v=t}^{u} (M - H_v)$. By (U), there are no holes in slot $u$, $H_u = 0$. Therefore,

$$R = \sum_{v=t}^{u} \left( \sum_{T \in \tau} \mathit{flow}(T, v) \right) - \sum_{v=t}^{u-1} (M - H_v) - M. \tag{4.33}$$

The right-hand side of (4.33) can be rewritten as follows.

$$\sum_{T \in \tau} (\mathit{flow}(T, t) + \mathit{flow}(T, u)) - (M - H_t) - M + \sum_{v=t+1}^{u-1} \left( \sum_{T \in \tau} \mathit{flow}(T, v) - (M - H_v) \right)$$

Rearranging terms, and using $\sum_{T \in I} \mathit{flow}(T, t) = 0$ (which follows by the definition of $I$), we get

$$\begin{aligned} \sum_{T \in B} \mathit{flow}(T, t) \quad &+ \sum_{T \in B \cup I_u \cup A_u} \mathit{flow}(T, u) - M + \sum_{T \in A_t} \mathit{flow}(T, t) - (M - H_t) \\ &+ \sum_{v=t+1}^{u-1} \left( \sum_{T \in \tau} \mathit{flow}(T, v) - (M - H_v) \right). \end{aligned}$$

By (4.30) – (4.32), the above value is non-positive. Hence, by (4.33), $LAG(\tau, u+1) - LAG(\tau, t) \leq 0$. Because $LAG(\tau, t) < 1$, this implies that $LAG(\tau, u+1) < 1$.

By (U) and (4.27), $t + 1 \leq u \leq \mathbf{min}(D(U_j), t_d) - 1$. Hence, $t + 2 \leq u + 1 \leq t_d$. Thus, there exists a $v \in \{t+2, \ldots, t_d\}$ such that $LAG(\tau, v) < 1$. ■

Recall our assumption that $t$ is the latest time such that $LAG(\tau, t) < 1$ and $LAG(\tau, t+1) \geq 1$. Because $t \leq t_d - 2$ (by (4.10)), we have $t + 2 \leq t_d$. By Lemma 4.14, $LAG(\tau, v) \leq 0$ for some $v \in \{t+2, \ldots, t_d\}$. By Lemma 4.8, parts (e) and (f), $v$ cannot be $t_d$ or $t_d - 1$. Thus, $v \leq t_d - 2$. Because $LAG(\tau, t_d) \geq 1$, this contradicts the maximality of $t$. Therefore, $t_d$ and $\tau$ as defined cannot exist. Thus, we have the following.

**Theorem 4.5.** PD$^2$ *correctly schedules any dynamic GIS task system satisfying* (J2) *and* (L2).

Since any feasible static IS task system satisfies (J2) and (L2), we have the following result.

**Corollary 4.5.1.** $\mathrm{PD}^2$ *is optimal for scheduling static IS task systems on multiprocessors.*

Further, since any sporadic task is also an IS task, we have the following corollary.

**Corollary 4.5.2.** $\mathrm{PD}^2$ *is optimal for scheduling static sporadic task systems on multiprocessors.*

## 4.4 Summary

In this chapter, we introduced a new task model called the intra-sporadic (IS) task model. The IS task model has a very flexible notion of a rate, and it generalizes the well-studied sporadic task model [Mok83]. We stated and proved the feasibility condition for scheduling IS tasks on multiprocessors. In addition, we also considered dynamic IS task systems, and provided sufficient conditions that ensure that no deadlines are missed under $\mathrm{PD}^2$. As a corollary, we obtained that $\mathrm{PD}^2$ is optimal for scheduling static IS task systems on multiprocessors.

# Chapter 5

# The Earliest-pseudo-deadline-First Algorithm*

In this chapter, we consider the problem of scheduling real-time applications on multiprocessors using the earliest-pseudo-deadline-first (EPDF) Pfair scheduling algorithm. As the name suggests, the EPDF algorithm prioritizes subtasks with earlier deadlines, and any ties between subtasks with equal deadlines is broken arbitrarily. (Thus, EPDF is essentially $PD^2$ without any of its tie-breaking rules.)

Recall from Chapter 1 that, in a hard real-time system, all task deadlines must be guaranteed, whereas occasional deadline misses or deadline misses by a small amount can be tolerated in a soft real-time system. Soft real-time systems have become very common with the proliferation of multimedia and gaming systems. While deadline misses are tolerated, they are obviously undesirable, and system quality and performance may be negatively impacted if tasks miss their deadlines either too often or by too much. One criterion used to measure system quality in the study of soft real-time systems is *tardiness*. If a job with a deadline at time $d$ completes at time $t$, then its tardiness is $\mathbf{max}(0, t - d)$. That is, if a job misses its deadline, then its tardiness indicates by how much. In this chapter, when considering soft real-time systems, we focus

*Most of the results presented in this chapter have been published in the following papers.

[SA04a] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. *Journal of Embedded Computing*, June 2004. Under submission. (A preliminary version of this paper was presented at the 15th Euromicro Conference on Real-time Systems [SA03].)

[SA04b] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 2004. Under submission. (A preliminary version of this paper was presented at the 11th International Workshop on Parallel and Distributed Real-time Systems.)

on the problem of minimizing tardiness.

Pfair scheduling algorithms such as $PD^2$ or PF, which are optimal for hard real-time systems, can be simplified (and hence, made more efficient) if deadline misses can be tolerated. Such simplified algorithms may be useful in a number of soft real-time applications implemented on multiprocessors. For example, consider a web server that services a large number of connections concurrently, some of which involve audio or video streaming that requires quality-of-service (QoS) guarantees. Such guarantees can be ensured by using fair scheduling disciplines. To handle a large number of connections, a multiprocessor platform may be necessary. Ensim Corp. has deployed fair multiprocessor scheduling algorithms in its product line for this very reason [CAGS00, CAS01].

The necessity of fair scheduling on multiple resources may also arise in the area of networking. Next-generation multimedia applications such as immersive reality systems will result in high bandwidth usage. One way to increase network bandwidth is to install multiple parallel links between pairs of connected routers. The problem of scheduling packets on outgoing links at a router then becomes a multiprocessor scheduling problem [ABJ99]. Fairness is desirable in this setting, just as it is in single-link scheduling [BZ96, DKS89, Gol94, PG93, Zha91]. Similar scheduling issues also arise in optical networks where wavelength-division-multiplexing (WDM) techniques are used to transmit multiple "light packets" at different wavelengths simultaneously [BR02].

As a final example, consider multiprocessor router platforms that process multiple packets simultaneously. The need for multiprocessor platforms in this context is necessitated by the growing disparity between link capacities and processor speeds [AHKB00, CO01]. Routers built using programmable network processors are destined to implement fairly complex packet-processing functions in software, making *processing* capacity (as opposed to *link* capacity) a critical resource to be managed [$SHA^+03$]. In this setting, fair scheduling is needed to ensure that QoS guarantees can be provided to different flows.

Note that in each of the above applications, an extreme degree of fairness that requires *all* deadlines to be met is not warranted. That is, these applications fall within the class of multiprocessor soft real-time systems. Further, note that all these systems are highly dynamic; the set of tasks or flows may change frequently. Hence, in this chapter, we assume that the task system under consideration is dynamic as well. We further assume that, instead of (J2) and (L2), the more liberal join and leave conditions (J1) and (L1) are used (refer to Section 4.2).

In Section 5.1, we consider the scheduling of hard real-time multiprocessor systems using EPDF. As a corollary of the main result, it follows that EPDF is optimal for two-processor systems, and three-processor systems consisting solely of light tasks. Later, in Section 5.2, we consider soft real-time systems.

## 5.1 Hard Real-time Systems

In this section, we show that task systems that are subject to the following condition can be correctly scheduled using EPDF on $M$ processors.

**(M0)** At any time, the sum of the weights of the $M-1$ heaviest tasks is at most one. Further, tasks are allowed to join and leave the system under the conditions (J1) and (L1).

The proof technique used here is very similar to the one used in Section 4.3.

We begin by making the assumption that there exists an task system $\tau$ that satisfies (M0) and yet misses a deadline under EPDF; we then show that this assumption leads to a contradiction. Let $S$ denote the EPDF schedule of $\tau$ in which a deadline is missed. Further, let $T_i$ be the subtask with the earliest deadline among all subtasks that miss a deadline, and let $t_d = d(T_i)$. Thus, all subtasks with deadlines less than $t_d$ meet their deadlines.

Note that any subtask with deadline after $t_d$ is scheduled in a slot prior to $t_d$ only if no subtask with a deadline at most $t_d$ is eligible in that slot. Thus, the scheduling of $T_i$ is not affected by subtasks with deadlines greater than $t_d$. Hence, a deadline is missed even if such subtasks are absent from the system. Therefore, we assume that no task in $\tau$ releases any subtask that has a deadline greater than $t_d$. In other words,

$$\text{for every subtask } U_j \in \tau, d(U_j) \leq t_d. \tag{5.1}$$

Using this, we obtain the following lower bound on $LAG(\tau, t_d)$.

**Lemma 5.1.** $LAG(\tau, t_d) \geq 1$.

*Proof.* By (4.9), we have

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

The first term on the right-hand side of the above equation is the total share in the ideal schedule over the interval $[0, t_d)$, which equals the total number of subtasks in

$\tau$ (since the deadlines of all subtasks in $\tau$ are at most $t_d$ by (5.1).) The second term corresponds to the number of subtasks scheduled in $[0, t_d)$ in $S$. Since $T_i$ misses its deadline at $t_d$, the difference between these two terms is at least one. ■

We now show that the same lower bound applies to $LAG(\tau, t_d - 1)$ as well. The following lemma is used to obtain that result.

**Lemma 5.2.** *There is no hole in slot $t_d - 1$.*

*Proof.* Let $U_j$ be any subtask that misses its deadline at $t_d$ in $S$. Because $d(U_j) = t_d$), $d(U_k) \leq t_d - 1$, where $U_k$ is $U_j$'s predecessor. By the minimality of $t_d$, $U_k$ meets its deadline and hence is scheduled before $t_d - 1$. Thus, if there were a hole in slot $t_d - 1$, then $U_j$ would have been scheduled there, in which case it would meet its deadline. Contradiction. ■

**Lemma 5.3.** $LAG(\tau, t_d - 1) \geq 1$.

*Proof.* By Lemma 5.2, there are no holes in slot $t_d - 1$. Hence, by Lemma 4.7, $LAG(\tau, t_d - 1) \geq LAG(\tau, t_d)$. Therefore, by Lemma 5.1, $LAG(\tau, t_d - 1) \geq 1$. ■

Because $LAG(\tau, 0) = 0$, it follows by Lemma 5.3 that there exists a time $t$ as follows.

$$(t < t_d - 1) \text{ and } (LAG(\tau, t) < 1) \text{ and } (LAG(\tau, t + 1) \geq 1) \tag{5.2}$$

We now prove some properties about task lags at time $t+1$; using these properties and (M0), we later derive a contradiction by showing that $(LAG(\tau, t + 1) < 1)$.

By Lemma 4.7, there is at least one hole in slot $t$ (*i.e.*, a processor is idle over $[t, t + 1)$). In other words, the number of tasks scheduled in slot $t$ is at most $M - 1$. Let $A$ denote the set of tasks scheduled in slot $t$. Then, we have

$$|A| \leq M - 1. \tag{5.3}$$

As in the proof for PD$^2$ in Section 4.3.3, we separate the rest of the tasks into two sets: $B$ and $I$. $B$ denotes the set of tasks not in $A$ that are active at time $t$. (Recall that a task $U$ is active at time $t$ if it has a subtask $U_j$ such that $e(U_j) \leq t < d(U_j)$.) We have the following result about tasks in set $B$.

**Claim 5.1.** *For any subtask $U_j$ of task $U \in B$, if $e(U_j) \leq t$, then $U_j$ is scheduled before $t$.*

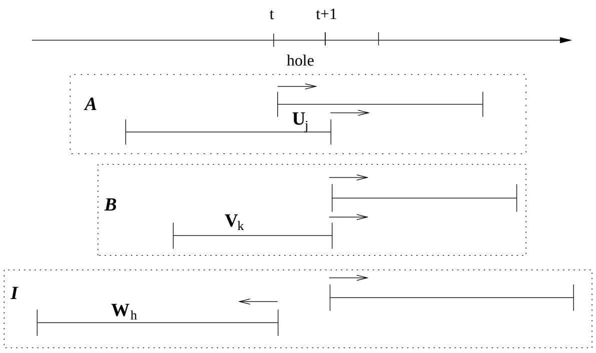



Figure 5.1: Sets $A$, $B$, and $I$. The PF-windows of a sample task of each set are shown.

*Proof.* If $d(U_j) \leq t$, then $d(U_j) < t_d$ (by (5.2)). By the minimality of $t_d$, $U_j$ meets its deadline. Hence, it is scheduled before $t$.

We now consider the case in which $d(U_j) \geq t+1$. Suppose subtask $U_j$ is scheduled at or after $t+1$. Because $U \in B$, no subtask of $U$ is scheduled in slot $t$. Therefore, there exists a subtask $U_k$ that satisfies **(i)** $e(U_k) \leq t < d(U_k)$, **(ii)** $U_k$ is scheduled at or after $t+1$, and **(iii)** $U_{k-1}$ (if it exists) is scheduled before $t$. This contradicts the greedy nature of the EPDF algorithm, which would have scheduled $U_k$ in slot $t$. ■

Let $I$ denote the set of the remaining tasks that are not active at time $t$. Figure 5.1 shows how the tasks in $A$, $B$, and $I$ are scheduled. We now provide upper bounds for the *lag* values at time $t+1$ for the tasks in each of $A$, $B$, and $I$.

**Lemma 5.4.** *For $W \in I$, $lag(W, t+1) = 0$.*

*Proof.* Because $t < t_d$ (by (5.2)), every subtask of task $W$ with a deadline at most $t$ meets its deadline in $S$. Therefore, if task $W$ leaves the system at or before time $t$, then by (L1), it does so with zero lag. In the rest of this proof, we assume that the task $W$ does not leave the system before time $t+1$. Consider any subtask $W_h$ of task $W$. We divide our analysis into two cases depending on the value of $e(W_h)$.

**Case 1: $\boldsymbol{e(W_h) \geq t+1}$.** In this case, we have $r(W_h) \geq t+1$. Therefore, by (4.6), $f(W_h, u) = 0$ for all slots $u \leq t < r(W_h)$. Hence, the share of $W_h$ over the interval $[0, t+1)$ in the ideal schedule is zero. Also, since $e(W_h) \geq t+1$, $W_h$ is scheduled at or after $t+1$ in schedule $S$. Therefore, the share of $W_h$ over $[0, t+1)$ is zero in $S$ as well.

**Case 2: $\boldsymbol{e(W_h) \leq t}$.** In this case, by the definition of $I$, $d(W_h) \leq t < t_d$. Since such a subtask meets its deadline in $S$, $W_h$ is scheduled in $[0, t)$. Therefore, its share over $[0, t+1)$ is one in $S$. Further, since its PF-window lies completely in $[0, t+1)$, its share is one in the ideal schedule as well.

Thus, under both cases, each subtask of $W$ receives equal shares in $S$ and the ideal schedule over $[0, t+1)$. Therefore, $lag(W, t+1) = 0$. ■

**Lemma 5.5.** *For $V \in B$, $lag(V, t+1) \leq 0$.*

*Proof.* Consider any subtask $V_k$ of task $V$. As in the proof of Lemma 5.4, we consider two cases.

**Case 1: $\boldsymbol{r(V_k) \geq t+1}$.** By (4.6), the share of $V_k$ in $[0, t+1)$ in the ideal schedule is zero.

**Case 2: $\boldsymbol{r(V_k) \leq t}$.** Since $e(V_k) \leq r(V_k)$, we have $e(V_k) \leq t$. Therefore, by Claim 5.1, $V_k$ is scheduled before $t$. Thus, the share of $V_k$ in $[0, t+1)$ is one in $S$, and at most one in the ideal schedule. ($d(V_k)$ may be greater than $t+1$, in which case a portion of $V_k$'s share in the ideal schedule is allocated after $t+1$.)

Thus, for any subtask of $V$, its share in $[0, t+1)$ in $S$ is at least its share in $[0, t+1)$ in the ideal schedule. Hence, $lag(V, t+1) \leq 0$. ■

**Lemma 5.6.** *For $U \in A$, $lag(U, t+1) < wt(U)$.*

*Proof.* Let $U_j$ be the subtask of $U$ scheduled at time $t$. Since $t+1 < t_d$ (by (5.2), $U_j$ meets its deadline. Therefore, $d(U_j) \geq t+1$. If $d(U_j) > t+1$, then by (4.1) – (4.3), $r(U_{j+1}) \geq t+1$. Reasoning exactly as in the proof of Lemma 5.5, we can show that $lag(U, t+1) \leq 0$.

In the rest of the proof, we assume that $d(U_j) = t+1$. If $U_{j+1}$ exists, then by (4.4), we have $r(U_{j+1}) \geq d(U_j) - 1$, *i.e.*, $r(U_{j+1}) \geq t$. We now divide the analysis into two cases.

**Case 1: $\boldsymbol{U_{j+1}}$ does not exist or $\boldsymbol{r(U_{j+1}) \geq t+1}$.** In this case, by (4.6), the share in the ideal schedule over $[0, t+1)$ for any subtask after $U_j$ is zero. Because $d(U_j) = t+1$, all subtasks before $U_j$ have deadlines in $[0, t+1)$. Further, all these subtasks meet their deadlines in $S$. Hence, each subtask of $U$ receives equal shares over $[0, t+1)$ in $S$ and the ideal schedule. This implies that $lag(U, t+1) = 0$.

**Case 2: $r(U_{j+1}) = t$.** In this case, because $d(U_j) = t+1$, we have $r(U_{j+1}) = d(U_j)-1$. Therefore, by (4.1) and (4.2), $\theta(U_{j+1})+\left\lfloor \frac{j}{wt(U)} \right\rfloor = \theta(U_j)+\left\lceil \frac{j}{wt(U)} \right\rceil -1$. Because $\theta(U_{j+1}) \geq \theta(U_j)$ (by (4.3), and $\lfloor x \rfloor \geq \lceil x \rceil -1$, we have $\theta(U_{j+1}) = \theta(U_j)$. Therefore, $\left\lfloor \frac{j}{wt(U)} \right\rfloor = \left\lceil \frac{j}{wt(U)} \right\rceil - 1$, which implies that $\left\lfloor \frac{j}{wt(U)} \right\rfloor < j/wt(U)$.

Because $U_j$ is scheduled in $[0, t+1)$ in $S$, the excess share of $U$ in the ideal schedule over $[0, t+1)$ is due to $f(U_{j+1}, t)$. Therefore, we have $lag(U, t+1) \leq f(U_{j+1}, t)$, *i.e.*, $lag(U, t+1) \leq f(U_{j+1}, r(U_{j+1}))$.

By (4.6), $f(U_{j+1}, r(U_{j+1})) = (\left\lfloor \frac{j}{wt(U)} \right\rfloor + 1) \cdot wt(U) - j$. Hence, $lag(U, t+1) \leq (\left\lfloor \frac{j}{wt(U)} \right\rfloor +1) \cdot wt(U) - j < (j/wt(U)+1) \cdot wt(U) - j$. Thus, $lag(U, t+1) < wt(U)$.

Thus, in all cases we have $lag(U, t+1) < wt(U)$. ■

Because $LAG(\tau, t+1) = \sum_{U \in A \cup B \cup I} lag(U, t+1)$, by Lemmas 5.4 – 5.6, $LAG(\tau, t+1) < \sum_{U \in A} wt(U)$. By (5.3), $|A| \leq M-1$. Therefore, by (M0), $LAG(\tau, t+1) < 1$, contradicting our assumption about $t$. Thus, we have the following theorem.

**Theorem 5.1.** EPDF *correctly schedules on $M$ processors every task system that satisfies* (M0).

Since any feasible *static* task system satisfies (J1) and (L1), we obtain the following result.

**Corollary 5.1.1.** EPDF *correctly schedules any feasible static IS task system on $M$ processors, if the sum of the weights of the $M-1$ heaviest tasks is at most one.*

Note that (M0) is trivially satisfied for $M = 1$ or $M = 2$, because the weight of any single task is at most one. Thus, we have the following result.

**Corollary 5.1.2.** EPDF *is optimal for scheduling IS tasks if number of processors is one or two.*

(This generalizes the result of Baruah [Bar95] that EPDF is optimal for scheduling periodic tasks on one processor.)

Ensuring (M0) involves identifying the $M-1$ heaviest tasks and adding up their weights. A more efficient (and more restrictive) way to enforce (M0) is to require each individual task weight to be at most $1/(M-1)$.

**Corollary 5.1.3.** EPDF *correctly schedules any dynamic IS task system satisfying* (J1) *and* (L1) *on* $M$ $(> 1)$ *processors if the weight of each task is at most* $1/(M-1)$.

As before, this implies the following result.

**Corollary 5.1.4.** EPDF *correctly schedules any feasible static IS task system in which the weight of each task is at most* $1/(M-1)$.

Substituting $M = 3$, we obtain the following.

**Corollary 5.1.5.** EPDF *correctly schedules any feasible IS task system on three processors if the weight of each task is at most* $1/2$.

### 5.1.1 Improving (M0)

Though the discussion above indicates that (M0) is reasonably tight, it can be slightly improved by more accurately bounding $lag(U, t+1)$ for $U \in A$, as shown below. We use the following result in our proof.

**Theorem 5.2 (Hardy *et al.* [HW79]).** *The smallest positive integer value of* $ax + by$ *is* $gcd(a, b)$ *for any integers* $a, b, x, y$.

Let $U.f = \dfrac{U.e - gcd(U.e, U.p)}{U.p}$, where $U.e$ and $U.p$ are $U$'s execution requirement and period, respectively. Then, we have the following.

**Lemma 5.7.** *If* $b(U_j) = 1$*, then* $f(U_{j+1}, r(U_{j+1})) \le U.f$.

*Proof.* By (2.11), $b(U_j) = 1$ implies that $\left\lceil \frac{j}{wt(U)} \right\rceil = \left\lfloor \frac{j}{wt(U)} \right\rfloor + 1$, which implies that $\left\lfloor \frac{j}{wt(U)} \right\rfloor < \frac{j}{wt(U)}$. Let $k = \left\lfloor \frac{j}{wt(U)} \right\rfloor$. Then, $\frac{j \cdot U.p}{U.e} - k > 0$, *i.e.*, $j \cdot U.p - k \cdot U.e > 0$. Therefore, by Theorem 5.2,

$$j \cdot U.p - k \cdot U.e \ge gcd(U.e, U.p). \tag{5.4}$$

Then, we have the following.

$$\begin{aligned} f(U_{j+1}, r(U_{j+1})) &= \left( \left\lfloor \frac{j}{wt(U)} \right\rfloor + 1 \right) \times wt(U) - j \quad \text{, by (4.6)} \\ &= \left( \left\lfloor \frac{j}{wt(U)} \right\rfloor + 1 \right) \times \frac{U.e}{U.p} - j \quad \text{, because } wt(U) = \tfrac{U.e}{U.p} \end{aligned}$$

$$
\begin{aligned}
&= (k+1) \times \frac{U.e}{U.p} - j && \text{, substituting } k = \left\lfloor \frac{j}{wt(U)} \right\rfloor \\
&= \frac{U.e}{U.p} - \frac{j \cdot U.p - k \cdot U.e}{U.p} && \text{, by simplification} \\
&\leq \frac{U.e}{U.p} - \frac{gcd(U.e, U.p)}{U.p} && \text{, by (5.4)} \\
&= U.f
\end{aligned}
$$

Thus, $f(U_{j+1}, r(U_{j+1})) \leq U.f$. ■

Using Lemma 5.7, we can improve the result in Lemma 5.6 to show that $lag(U, t+1) \leq U.f$ for all $U \in A$. Performing the same analysis as before, we obtain a contradiction if $\sum_{U \in A} U.f < 1$. Thus, we have the following theorem.

**Theorem 5.3.** EPDF *correctly schedules a dynamic IS task system* $\tau$ *that satisfies* (J1) *and* (L1) *if, at all times,* $\sum_{U \in H} U.f < 1$ *for all sets* $H \subseteq \tau$ *of at most* $M-1$ *tasks.*

### 5.1.2 Tightness

We now prove that the sufficiency condition presented in Theorem 5.3 is tight; in particular, we show the following.

**Theorem 5.4.** *There exists a feasible static periodic task system* $\tau$ *that misses a deadline on* $M$ $(> 2)$ *processors under* EPDF *if* $\sum_{T \in H} T.f$ *is allowed to be at least* 1 *for some set* $H \subseteq \tau$ *of at most* $M-1$ *tasks.*

*Proof.* Let $\tau$ be the following system of $(M-1)^2$ tasks: set $A$ consisting of $M-1$ tasks of weight $\frac{1}{M-1} + \frac{1}{(M-1)^2}$ $\left(= \frac{M}{(M-1)^2}\right)$ and set $B$ consisting of $M(M-2)$ tasks of weight $\frac{1}{M-1}$. (Figure 5.2 illustrates the case for $M = 5$.) Note that $\tau$ fully utilizes the $M$ processors because $(M-1) \times \frac{M}{(M-1)^2} + \frac{M(M-2)}{M-1} = \frac{M + M(M-2)}{M-1}$, which simplifies to $\frac{M^2 - M}{M-1} = M$. Further note that for each task $T \in A$, $T.f = \frac{M-1}{(M-1)^2}$ because $gcd(M, (M-1)^2) = 1$. Thus, $\sum_{T \in A} T.f = (M-1) \times \left(\frac{1}{M-1}\right) = 1$.

We first show that for all tasks $T \in \tau$, $d(T_1) = M-1$. If $T \in B$, by (2.8), we have $d(T_1) = \lceil M-1 \rceil = M-1$. If $T \in A$, by (2.8), $d(T_1) = \left\lceil \frac{(M-1)^2}{M} \right\rceil$. Note that $\frac{(M-1)^2}{M} = \frac{M^2 - 2M + 1}{M} = M - 2 + \frac{1}{M}$. Therefore, $d(T_1) = \left\lceil M - 2 + \frac{1}{M} \right\rceil = M - 1$.

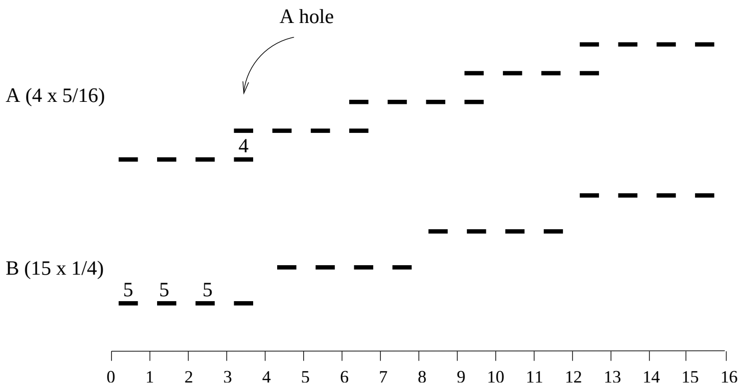



Figure 5.2: Tightness of Theorem 5.3. A partial EPDF schedule is depicted for four tasks of weight 5/16 and 15 tasks of weight 1/4 on 5 processors. There is a hole in slot 4; since the total utilization is 5, this implies a deadline miss in the future.

Thus, EPDF may assign higher priority to all the tasks in set $B$ at time 0, thereby scheduling them for the first $M-2$ slots. The tasks in set $A$ will be scheduled in slot $M-2$ (*i.e.*, the interval $[M-2, M-1)$). Because $r(T_2) = \lfloor M-1 \rfloor = M-1$ for all $T \in B$, no other tasks are eligible to be scheduled in slot $M-2$. Since $A$ has only $M-1$ tasks, a processor will be idle in slot $M-2$. Because $\tau$ fully utilizes $M$ processors, this implies that a deadline is missed in the future. ■

### 5.1.3 Other Schedulability Results

In Theorems 5.1 and 5.3, we presented restrictions on individual task weights that ensure that EPDF does not miss any deadlines. We now present different schedulability tests that do not place restrictions on individual task weights; however, the pseudo-deadlines used in the EPDF algorithm are slightly different. We base our results on the following theorem.

**Theorem 5.5.** EPDF *correctly schedules any feasible IS task system if the weight of each task is a reciprocal of some integer.*

*Proof.* Let $\tau$ be a feasible IS task system $\tau$ in which the weight of each task is a reciprocal of some integer, *i.e.*, for each task $T \in \tau$, $wt(T) = \frac{1}{k}$ for some integer $k$. We show that, for such a task system, the $b$-bit and the group deadline can be eliminated from the PD$^2$ priority definition without any change in its scheduling decisions. If $wt(T) = \frac{1}{k}$, then by (4.2), we have $d(T_i) = \theta(T_i) + ik$ for all $i$.

Also, by (2.11), $b(T_i) = \lceil ik \rceil - \lfloor ik \rfloor = 0$ for all $i$. Thus, the $b$-bit is zero for all subtasks of each task in $\tau$. Further, by (4.7), $D(T_i) = \theta(T_i) + \left\lceil \frac{\lceil ik \times (k-1)/k \rceil}{(k-1)/k} \right\rceil$. This simplifies to $\theta(T_i) + \left\lceil \frac{i(k-1)}{(k-1)/k} \right\rceil = \theta(T_i) + \lceil ik \rceil$. Hence, $D(T_i) = \theta(T_i) + ik = d(T_i)$.

Thus, if $d(T_i) = d(U_j)$, then $b(T_i) = b(U_j)$ and $D(T_i) = D(U_j)$. This implies that $\text{PD}^2$ breaks ties between equal subtask deadlines in an arbitrary manner, and hence behaves identically to EPDF. It follows by Corollary 4.5.1 that EPDF correctly schedules $\tau$. ■

Now, given any task system $\tau$, we can transform it into a new task system $\tau'$ that satisfies the conditions of Theorem 5.5. We modify the parameters of every task in $\tau$ to obtain $\tau'$ as follows. For every task $T \in \tau$, we construct a task $T' \in \tau'$ with weight $\frac{1}{\lfloor 1/wt(T) \rfloor}$. The parameters of each subtask $T'_i$ of $T'$ is obtained from those of subtask $T_i$ as follows.

**(a)** $e(T'_i) = e(T_i)$,

**(b)** $r(T'_i) = r(T_i)$,

**(c)** $\theta(T'_i) = r(T'_i) - \left\lfloor \frac{i-1}{wt(T')} \right\rfloor = r(T'_i) - \left\lfloor (i-1) \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor \right\rfloor = r(T'_i) - (i-1) \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor$, and

**(d)** $d(T'_i) = \theta(T'_i) + \left\lceil \frac{i}{wt(T')} \right\rceil = \theta(T'_i) + \left\lceil i \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor \right\rceil = \theta(T'_i) + i \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor$.

Thus, the window of $T'_i$ is set to start at the same time as $T_i$'s window. Figure 5.3 illustrates this for a task of weight 3/10. The scheduler schedules $\tau$ by effectively scheduling $\tau'$ using EPDF and selecting $T_i$ for execution whenever EPDF selects $T'_i$.

By (c) and (d), $d(T'_i) = r(T'_i) + \left\lfloor \frac{1}{wt(T)} \right\rfloor$, and by (4.1) and (4.2), $d(T_i) = r(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$. Thus, by (b), we have the following.

$$\begin{aligned}
d(T_i) - d(T'_i) &= \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor - \left\lfloor \frac{1}{wt(T)} \right\rfloor \\
&\geq \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i}{wt(T)} \right\rfloor \qquad , \quad \lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor \\
&\geq 0 \qquad , \quad \lceil x \rceil \geq \lceil x \rceil
\end{aligned}$$

Thus, $d(T'_i) \leq d(T_i)$. Therefore, if $T'_i$ meets its deadline, then $T_i$ also meets its deadline. By Theorem 5.5, EPDF correctly schedules $\tau'$ on $M$ processors if $\sum_{T' \in \tau'} wt(T') \leq M$. Thus, we have the following theorem.

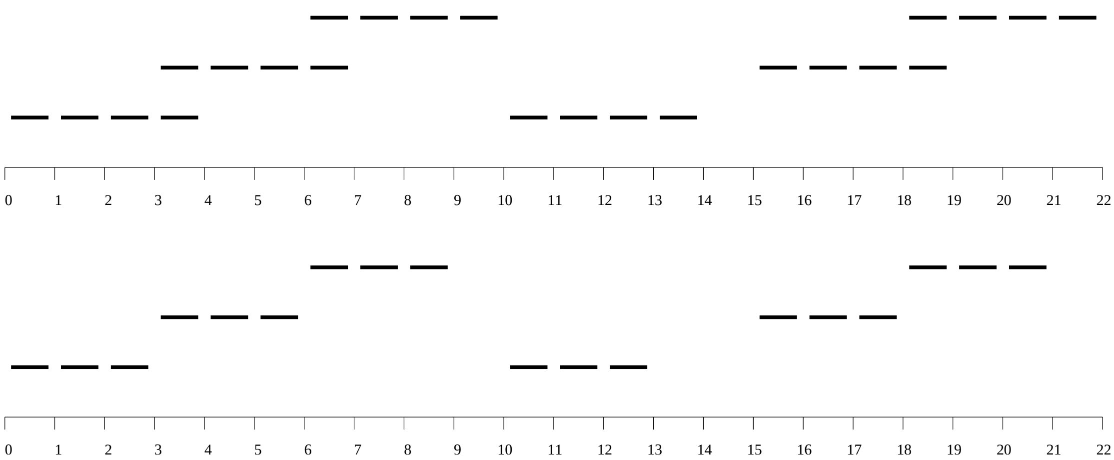



Figure 5.3: Example illustrating transformation of a task $T$ of weight $3/10$ into a task $T'$ of weight $\frac{1}{\lfloor 10/3 \rfloor} = 1/3$. The subtask windows of $T'$ are placed so that the release times of its subtasks coincide with those of $T$'s subtasks.

**Theorem 5.6.** EPDF (*with deadlines determined by* (*d*)) *correctly schedules* $\tau$ *on* $M$ *processors if* $\sum_{T\in\tau} \frac{1}{\lfloor 1/wt(T) \rfloor} \leq M$.

The following results follow as simple corollaries to Theorem 5.6.

**Corollary 5.6.1.** EPDF (*with deadlines determined by* (*d*)) *correctly schedules* $\tau$ *on* $M$ *processors if* $\sum_{T\in\tau} \frac{wt(T)}{1-wt(T)} \leq M$.

*Proof.* Note that $\left\lfloor \frac{1}{wt(T)} \right\rfloor > \frac{1}{wt(T)} - 1$ , *i.e.*, $\left\lfloor \frac{1}{wt(T)} \right\rfloor > \frac{1-wt(T)}{wt(T)}$. Therefore, $\frac{1}{\lfloor 1/wt(T) \rfloor} < \frac{wt(T)}{1-wt(T)}$. The required result follows by Theorem 5.6. ■

**Corollary 5.6.2.** EPDF (*with deadlines determined by* (*d*)) *correctly schedules* $\tau$ *on* $M$ *processors if* $\sum_{T\in\tau} wt(T) \leq M/2$.

*Proof.* We show that $2 \cdot wt(T) \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor > 1$, which implies that $\frac{1}{\lfloor 1/wt(T) \rfloor} < 2 \cdot wt(T)$. The required result then follows by Theorem 5.6. Let $k \geq 1$ be such that $\frac{1}{k+1} < wt(T) \leq \frac{1}{k}$. Then, $\frac{1}{wt(T)} \geq k$. Since $k$ is an integer, this implies that $\left\lfloor \frac{1}{wt(T)} \right\rfloor \geq k$. Because $wt(T) > \frac{1}{k+1}$, we obtain $2 \cdot wt(T) \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor > \frac{2k}{k+1}$. Note that $2k \geq k+1$ because $k \geq 1$. Thus, $2 \cdot wt(T) \cdot \left\lfloor \frac{1}{wt(T)} \right\rfloor > 1$. ■

The above-described schedulability tests are useful in hard real-time systems in which some tasks have weights more than $\frac{1}{(M-1)}$, and the task system as a whole does not fully utilize all processors.

## 5.2 Soft Real-time Systems

In this section, we consider soft real-time multiprocessor systems, in which minimizing tardiness is the primary concern. The notion of tardiness for jobs was defined earlier in the beginning of this chapter. *Tardiness of a subtask* is defined similarly: if $t$ is the time at which subtask $T_i$ completes, then $\mathbf{max}(0, t - d(T_i))$ is its tardiness. The *tardiness of a task system* is defined as the maximum tardiness among all of its subtasks in any schedule. We now present a condition on the weights of tasks in a system that is sufficient for ensuring that EPDF maintains a tardiness of at most one quantum. Later, in Section 5.2.2, we generalize this condition for tardiness thresholds that exceed one.

### 5.2.1 Sufficient Condition for Tardiness of at most One

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

**(M1)** At all times, the sum of the $M - 1$ largest task weights is at most $\frac{M+1}{2}$. Further, tasks are allowed to join and leave the system under the conditions (J1) and (L1).

As in Section 5.1, we begin with the following assumption: there exists a feasible task system $\tau$ satisfying (M1) with tardiness greater than one. Let $S$ denote the EPDF schedule of $\tau$ in which some subtask has a tardiness greater than one. Further, let $T_i$ be the subtask with the earliest deadline among all subtasks that have tardiness greater than one, and let $t_d = d(T_i)$. Thus, all subtasks with deadlines less than $t_d$ have tardiness at most one.

Under EPDF, since the scheduling of $T_i$ is not affected by subtasks with deadlines greater than $t_d$, we assume that no task in $\tau$ releases any subtask with a deadline greater than $t_d$. In other words,

$$\text{for every subtask } U_j \in \tau, d(U_j) \leq t_d. \tag{5.5}$$

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

**Lemma 5.8.** *At least $M+1$ subtasks in $\tau$ miss their deadlines at $t_d$.*

*Proof.* Consider any subtask $U_j$ such that $U_j$ misses its deadline at $t_d$. Because $d(U_j) = t_d$, by (4.2) and (4.3), the deadline of $U_{j-1}$ is at or before $t_d - 1$. Therefore, by the definition of $T_i$ and $t_d$, $U_{j-1}$ has tardiness at most one and is scheduled in $[0, t_d)$. Hence, $U_j$ is eligible to be scheduled at time $t_d$. If there are at most $M$ subtasks like $U_j$ (including $T_i$), then each of these subtasks will be scheduled in $[t_d, t_d+1)$. Therefore, subtask $T_i$ cannot have tardiness greater than one. Contradiction. ■

**Lemma 5.9.** $LAG(\tau, t_d) \geq M+1$.

*Proof.* By (4.9), we have

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

The first term on the right-hand side of the above equation is the total share in the ideal schedule in $[0, t_d)$, which equals the total number of subtasks in $\tau$. (This follows from (5.5), and because $\tau$ is feasible.) The second term corresponds to the number of subtasks scheduled over $[0, t_d)$ in $S$. Since at least $M+1$ subtasks miss their deadlines at $t_d$ (by Lemma 5.8), the difference between these two terms is at least M+1. ■

Similar to the results in Lemmas 5.2 and 5.3, we can use Lemma 5.9 to show that $LAG(\tau, t_d - 1) \geq M+1$. Because $LAG(\tau, 0) = 0$, it follows that there exists a time $t < t_d$ as follows.

$$t < t_d - 1 \text{ and } LAG(\tau, t) < M+1 \text{ and } LAG(\tau, t+1) \geq M+1. \tag{5.6}$$

We now prove some properties about task lags at time $t+1$; using these properties and (M1), we later derive a contradiction concerning the existence of $t$.

We define the sets $A$, $B$, and $I$ as in Section 5.1. Thus, set $A$ satisfies (5.3). Figure 5.4(a) shows how the tasks in $A$, $B$, and $I$ are scheduled. We now provide upper bounds for the $lag$ values at time $t+1$ for the tasks in each of $A$, $B$, and $I$.

**Lemma 5.10.** *For $W \in I$, $lag(W, t+1) = 0$.*

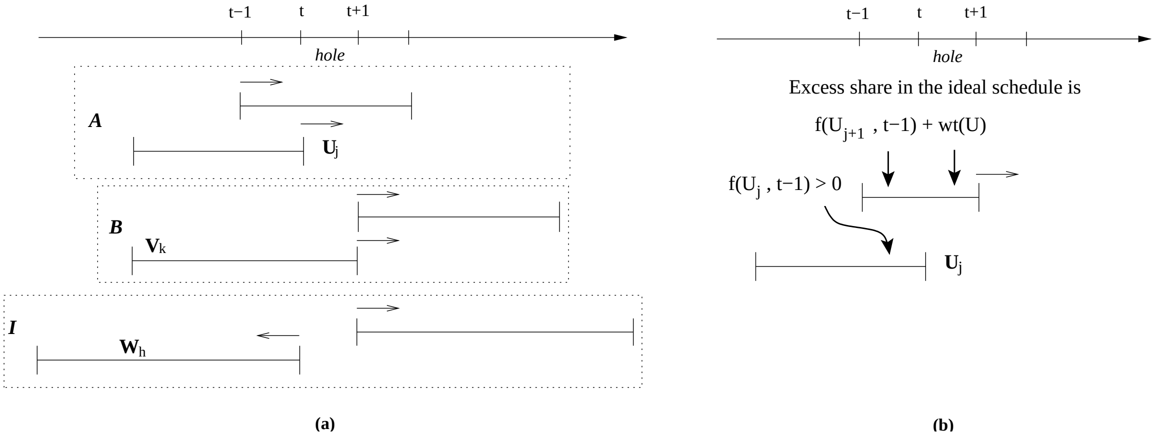



Figure 5.4: In this figure, PF-windows are denoted by line segments. An arrow over a release (respectively, deadline) indicates that the release (respectively, deadline) could be anywhere in the direction of the arrow. There is a hole in slot $t$. **(a)** Sets $A$, $B$, and $I$. The PF-windows of a sample task of each set are shown. **(b)** Case 4 of Lemma 5.12. The excess share received by $U$ in $[0, t+1)$ in the ideal schedule is equal to the shares received by subtasks after $U_j$ in $[0, t+1)$. (Note that if $U_{j+1}$ has a deadline at time $t+1$, then the PF-window of $U_{j+2}$ may begin at time $t$, in which case part of the $wt(U)$ share in slot $t$ is due to $U_{j+2}$.)

*Proof.* If task $W$ leaves the system at or before time $t$, then by (L1) (refer to Section 4.2) it leaves with zero lag. (If $W$ releases a subtask $W_h$ and leaves before $W_h$ gets scheduled, then it is equivalent to $W$ not releasing $W_h$ at all.) Therefore, we assume that $W$ does not leave the system before time $t+1$. Consider any subtask $W_h$ of task $W$. We consider two cases depending on the value of $e(W_h)$.

**Case 1: $\boldsymbol{e(W_h) \geq t+1}$.** In this case, we have $r(W_h) \geq t+1$. Therefore, by (4.6), $f(W_h, u) = 0$ for all slots $u \leq t < r(W_h)$. Hence, the share of $W_h$ in the ideal schedule in $[0, t+1)$ is zero. Also, since $e(W_h) \geq t+1$, $W_h$ is scheduled at or after $t+1$ in $S$. Therefore, the share of $W_h$ over $[0, t+1)$ in $S$ is zero as well.

**Case 2: $\boldsymbol{e(W_h) \leq t}$.** In this case, by the definition of $I$, $d(W_h) \leq t < t_d$. Because the tardiness of such a subtask is at most one, $W_h$ is scheduled in $[0, t+1)$. Hence, the share received by $W_h$ in $[0, t+1)$ is one in both the ideal and EPDF schedules.

Thus, under both cases, each subtask of $W$ receives equal shares in $S$ and the ideal schedule over $[0, t+1)$. Therefore, $lag(W, t+1) = 0$. ■

**Lemma 5.11.** *For $V \in B$, $lag(V, t+1) \leq 0$.*

*Proof.* Consider any subtask $V_k$ of task $V$. Again, as in the proof of Lemma 5.10, we consider two cases.

**Case 1: $\boldsymbol{r(V_k) \geq t+1}$.** In this case, by (4.6), the share of $V_k$ in $[0, t+1)$ in the ideal schedule is zero.

**Case 2: $\boldsymbol{r(V_k) \leq t}$.** By Claim 5.1 (proved in Section 5.1), $V_k$ is scheduled before $t$. Thus, the share of $V_k$ in $[0, t+1)$ is one in $S$ (even if it has missed its deadline), and at most one in the ideal schedule. (If $d(V_k)$ is greater than $t+1$, a portion of $V_k$'s share in the ideal schedule is allocated after $t+1$.)

Thus, for any subtask of $V$, its share in $[0, t+1)$ in $S$ is at least its share in $[0, t+1)$ in the ideal schedule. Hence, $lag(V, t+1) \leq 0$. ∎

**Lemma 5.12.** *For $U \in A$, $lag(U, t+1) < 2 \times wt(U)$.*

*Proof.* Let $U_j$ be the subtask of $U$ scheduled at time $t$. We consider several cases depending on the value of $d(U_j)$.

We first show that $d(U_j) \geq t$. If $d(U_j) \leq t$, then because $t < t_d$, we have $d(U_j) < t_d$. By the choice of $T_i$ and $t_d$, it follows that the tardiness of $U_j$ is at most one. Since $U_j$ is scheduled in slot $t$, this implies that $d(U_j) \geq t$. Hence, in this case, $d(U_j) = t$. Thus, we have $d(U_j) \geq t$. We now consider four cases.

**Case 1: $\boldsymbol{d(U_j) > t+1}$.** In this case, by (4.4), $r(U_{j+1}) \geq t+1$. Reasoning exactly as in the proof of Lemma 5.11, we can show that $lag(U, t+1) \leq 0$.

**Case 2: $\boldsymbol{d(U_j) = t+1 \wedge b(U_j) = 0}$.** By (2.11), $b(U_j) = 0$ implies that $\left\lceil \frac{j}{wt(U)} \right\rceil = \left\lfloor \frac{j}{wt(U)} \right\rfloor$, and by (4.3), $\theta(U_{j+1}) \geq \theta(U_j)$. Therefore, by (4.1) and (4.2), $r(U_{j+1}) \geq t+1$. It follows that the share received by any subtask $U_k$ $(k > j)$ in the ideal schedule in $[0, t+1)$ is zero. Thus, the share received by task $U$ in $[0, t+1)$ is the same in both the ideal and EDPF schedules. Therefore, $lag(U, t+1) = 0$.

**Case 3: $\boldsymbol{d(U_j) = t+1 \wedge b(U_j) = 1}$.** (The proof of this case is similar to Case 2 in the proof of Lemma 5.6.) By (4.4), we have

$$r(U_{j+1}) \geq d(U_j) - 1. \tag{5.7}$$

Therefore, $r(U_{j+1}) \geq t$. By (4.1) – (4.3), we obtain $r(U_{j+2}) \geq t+1$. Therefore, the share of any subtask $U_k$ $(k > j+1)$ is zero in $[0, t+1)$ in the ideal schedule.

Now, the total share that $U$ receives in $[0, t+1)$ in $S$ schedule is $j$. However, in the ideal schedule, the share received by $U$ in $[0, t+1)$ may be more because of the share that $U_{j+1}$ (if it exists) receives in $[0, t+1)$. This share will be non-zero only if $r(U_{j+1}) \leq t$, *i.e.*, $r(U_{j+1}) \leq d(U_j) - 1$. In this case, by (5.7), we have $r(U_{j+1}) = d(U_j) - 1$. This implies that $\theta(U_{j+1}) = \theta(U_j)$. It also implies that the excess share in the ideal schedule in $[0, t+1)$ is at most $f(U_{j+1}, t)$. Because $r(U_{j+1}) = t$, by (4.6), we have $f(U_{j+1}, t) = (\left\lfloor \frac{j}{wt(U)} \right\rfloor + 1) \cdot wt(U) - j$.

Because $\theta(U_{j+1}) = \theta(U_j)$, by (4.1) and (4.2), $\left\lfloor \frac{j}{wt(U)} \right\rfloor = \left\lceil \frac{j}{wt(U)} \right\rceil - 1$. Hence, $\left\lfloor \frac{j}{wt(U)} \right\rfloor < j/wt(U)$. Therefore, $f(U_{j+1}, t) < (j/wt(U) + 1) \cdot wt(U) - j$, *i.e.*, $f(U_{j+1}, t) < wt(U)$. Thus, $lag(U, t+1) < wt(U)$.

**Case 4: $\boldsymbol{d(U_j) = t}$.** (In this case, $U_j$ misses its deadline.) By (4.4), we have $r(U_{j+1}) \geq t - 1$. Hence, by (4.1) – (4.3), $r(U_{j+2}) \geq t$. Since $U_j$ is scheduled in slot $t$, all subtasks of $U$ up to and including $U_j$ receive a share of one over $[0, t+1)$ in both $S$ and the ideal schedule (see Figure 5.4(a)).

By (4.6), $f(U_k, t) > 0$ only if $t \geq r(U_k)$. Therefore, the only subtask after $U_j$ that may contribute to $U$'s share in the ideal schedule in $[0, t)$ is $U_{j+1}$. Further, by (F1), the share of $U$ in slot $t$ (*i.e.*, in $[t, t+1)$) is at most $wt(U)$. Thus, it follows that the excess share that $U$ can receive in $[0, t+1)$ in the ideal schedule is at most $f(U_{j+1}, t-1) + wt(U)$ (refer to Figure 5.4(b)). The first term, $f(U_{j+1}, t-1)$, will be non-zero only if $r(U_{j+1}) = t-1$, *i.e.*, $r(U_{j+1}) = d(U_j) - 1$. Reasoning exactly as in Case 3, it follows that $f(U_{j+1}, t-1) < wt(U)$. Hence, $lag(U, t+1) < 2 \times wt(U)$.

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

Because $LAG(\tau, t+1) = \sum_{U \in A \cup B \cup I} lag(U, t+1)$, by Lemmas 5.10 – 5.12, $LAG(\tau, t+1) < 2 \times \sum_{U \in A} wt(U)$. By (5.3), $|A| \leq M - 1$. Therefore, by (M1), $LAG(\tau, t+1) < M + 1$, contradicting our assumption about $t$. Thus, we have the following theorem.

**Theorem 5.7.** *Every feasible IS task system satisfying* (M1) *has a tardiness of at most one under* EPDF.

One way to ensure (M1) is to restrict the weight of every task to at most $\frac{M+1}{2M-2}$.

**Corollary 5.7.1.** EPDF *guarantees a tardiness of at most one on $M$ processors for any dynamic IS task system satisfying* (J1) *and* (L1) *if the weight of each task is at most* $\frac{M+1}{2M-2}$.

Note that $\frac{M+1}{2M-2} = \frac{M-1+2}{2(M-1)} > \frac{1}{2}$. Thus, we have the following result.

**Corollary 5.7.2.** EPDF *guarantees a tardiness of at most one on $M$ processors for any dynamic IS task system satisfying* (J1) *and* (L1) *if the weight of each task is at most* $1/2$.

In other words, EPDF guarantees a tardiness of at most one for a task system consisting solely of light tasks.

**Discussion.** It is possible to improve (M1) by more accurately bounding the lag values for tasks in set $A$. In particular, as we show in Lemma 5.14 below, there must be at least one task $U \in A$ such that $d(U_j) \geq t+1$, where $U_j$ is the subtask scheduled in slot $t$. Using this, we prove that the following holds.

**Lemma 5.13.** *Let $\tau$ be a task system that satisfies* (J1) *and* (L1) *and let $w_1, w_2, \ldots$, $w_{M-1}$ denote the weights* (*in non-increasing order*) *of the $M-1$ heaviest tasks in $\tau$. If $w_{M-1} + 2 \times \sum_{i=1}^{M-2} w_i \leq M+1$, then* EPDF *guarantees a tardiness of at most one for $\tau$.*

This allows us to improve the individual weight restriction to $\frac{M+1}{2M-3}$. If every task has weight at most $\frac{M+1}{2M-3}$, then

$$\begin{aligned} w_{M-1} + 2 \times \sum_{i=1}^{M-2} w_i &\leq \frac{M+1}{2M-3} + \frac{2(M-2)(M+1)}{2M-3} \\ &= (M+1) \times \left(\frac{1+2M-4}{2M-3}\right) \\ &= M+1 \end{aligned}$$

Thus, we have the following result.

**Corollary 5.7.3.** EPDF *guarantees a tardiness of at most one on $M$ processors for any dynamic IS task system satisfying* (J1) *and* (L1) *if the weight of each task is at most* $\frac{M+1}{2M-3}$.

Note that, for $M \leq 4$, we have $2M-3 \leq M+1$, *i.e.*, $\frac{M+1}{2M-3} \geq 1$. Because the weight of any task is at most one, the individual weight restriction is always satisfied if $M \leq 4$. That is, for $M \leq 4$, EPDF guarantees a tardiness of at most one without any weight restriction.

We now prove that Lemma 5.13 applies to dynamic GIS task systems as well. This generalization allows us to use Lemmas 4.4 – 4.6 in our proof. As in the proof of Theorem 4.5, we consider a time $t_d$ and a task system instance $\tau$ defined as follows.

**Definition 5.1.** *$t_d$ is the earliest time that corresponds to the deadline of a subtask with tardiness two when any task system is scheduled using* EPDF. ■

**Definition 5.2.** *$\tau$ is an instance of a task system with the following properties.*

**(T1)** *$\tau$ has a subtask $T_i$ such that $d(T_i) = t_d$, and $T_i$ has a tardiness of two when $\tau$ is scheduled using* EPDF*.*

**(T2)** *No instance of any task system that satisfies* (T1) *releases fewer subtasks in* $[0, t_d)$ *than $\tau$.* ■

Let $S$ denote the EPDF schedule of $\tau$ in which subtask $T_i$ has a tardiness of two. Define sets $A$, $B$, and $I$ as in Section 5.1. Note that Lemma 4.9 applies here as well.

**Lemma 5.14.** *There exists a task $U \in A$ such that $d(U_j) \geq t + 1$, where $U_j$ is the subtask of $U$ scheduled in slot $t$.*

*Proof.* Because $t + 1 < t_d$ (by (5.6), for any subtask $U_j$ scheduled in slot $t$, its tardiness is at most one. Therefore, its deadline is at or after $t$. We now show that the following assumption leads to a contradiction.

**(A)** For all tasks $U \in A$, if $U_j$ is the subtask scheduled in slot $t$, then $d(U_j) = t$.

Consider any task $V \in B$. (Such a $V$ exists because $B$ is non-empty (by Lemma 4.9).) By the definition of set $B$, there exists a subtask $V_k$ such that $e(V_k) \geq t < d(V_k)$. We now show below that under assumption (A), $V_k$ can be removed without affecting the scheduling of subtasks after slot $t$.

**Claim 5.2.** *Under assumption* (A)*, $V_k$ can be removed without affecting the schedule after slot $t$.*

*Proof.* By Claim 5.1, $V_k$ must be scheduled before $t$. Let the chain of displacements caused by removing $V_k$ be $\Delta_1, \Delta_2, \ldots, \Delta_j$, where $\Delta_i = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \rangle$ and $X^{(1)} = U_j$. By Lemma 4.4, $t_{i+1} > t_i$ for $1 \leq i \leq k$.

Note that at slot $t_i$, the priority of subtask $X^{(i)}$ is at least that of $X^{(i+1)}$, because $X^{(i)}$ was chosen over $X^{(i+1)}$ in $S$. Thus, because $X^{(1)} = V_k$ and $d(V_k) \geq t+1$, we have the following.

$$\text{For all } i \in \{1, \ldots, k+1\},\ d(X^{(i)}) \geq t+1. \tag{5.8}$$

We now show that the displacements do not extend beyond slot $t$. Assume, to the contrary, that $t_{k+1} > t$. Consider $h \in \{2, \ldots, k+1\}$ such that $t_h > t$ and $t_{h-1} \leq t$. Such an $h$ exists because $t_1 < t < t_{k+1}$. Because there is a hole in slot $t$ and $t_{h-1} \leq t < t_h$, by Lemma 4.6, $t_{h-1} = t$. Therefore, $X^{(h-1)}$ is scheduled in slot $t$. In other words, $X^{(h-1)} \in A$. By (5.8), $d(X^{(h-1)}) \geq t+1$. This contradicts assumption (A). ■

Hence, under assumption (A), $V_k$'s removal does not affect the tardiness of of $T_i$ and $\tau$. Let $\tau'$ be the task system instance obtained by removal of subtask $V_k$. Then, $\tau'$ satisfies (T1) and has one fewer subtask than $\tau$. This contradicts (T2), implying that assumption (A) is false. ■

Let $U$ be the task in $A$ such that $d(U_j) \geq t+1$, where $U_j$ is the subtask of $U$ scheduled in slot $t$. Therefore, one of the first three cases in the proof of Lemma 5.12 applies to task $U$, and hence, $lag(U, t+1) < wt(U)$. If $A'$ is the set $A - \{U\}$, then $LAG(\tau, t+1) < wt(U) + 2 \times \sum_{V \in A'} wt(V)$. Therefore, if the condition in the statement of Lemma 5.13 is satisfied, then $LAG(\tau, t+1) < M+1$, which is a contradiction. Thus, $\tau$ as defined above cannot exist, implying Lemma 5.13.

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

Consider the following set of periodic tasks: three tasks of weight $\frac{1}{2}$ and $M-1$ tasks of weight $\frac{2M-3}{2M-2}$. Figure 5.5 illustrates the case $M = 5$. In these examples, the number of simultaneous deadline misses increases up to some multiple of the least common multiple of the task periods, and then remains steady after that. Note that, in these examples, the percentage of jobs that miss their deadlines is quite high (approaches 25% for large $M$). However, as our simulation experiments (described in Section 5.2.3) indicate, such cases are very rare indeed.

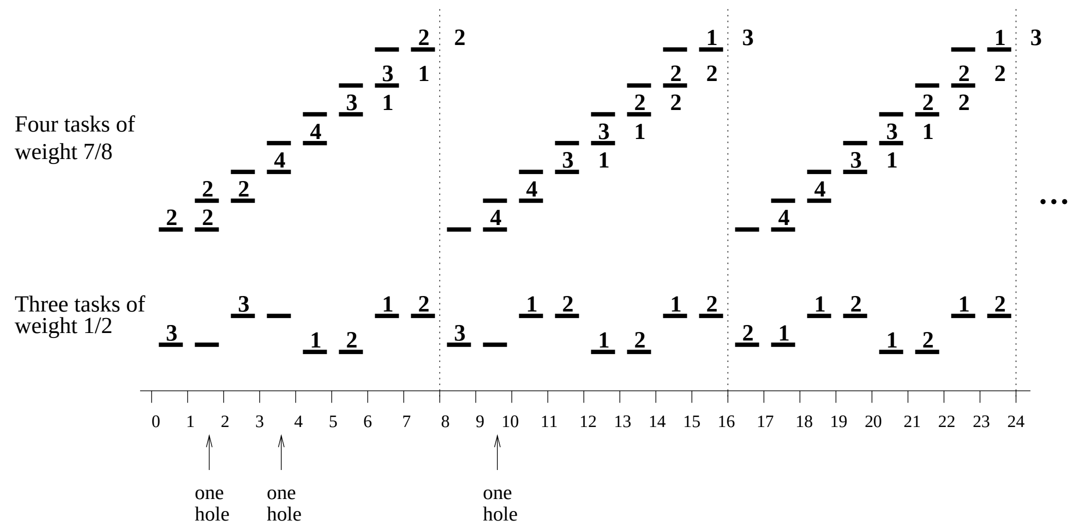



Figure 5.5: The Pfair window of each subtask is shown on a separate line. An integer value $n$ in slot $t$ means that $n$ of the corresponding subtasks are scheduled in slot $t$. No integer value means that no such subtask is scheduled in slot $t$. Each subtask's IS-window is same as its PF-window. Subtasks that miss their deadlines are shown scheduled after their windows. In the schedule shown, EPDF breaks ties in favor of the tasks of weight 1/2. Note that the schedule over $[16, 24]$ repeats after time 24. Thus, a maximum of three subtasks miss their deadlines simultaneously, and hence, the tardiness of each subtask is at most one.

### 5.2.2 Sufficient Condition for Tardiness of at most $k$

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

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

The proof is very similar to the sufficiency proof for (M1). $t_d$, $T_i$, and $\tau$ are defined in the same manner, except that the tardiness of $T_i$ is $k+1$. Similar to Lemma 5.8, we can show that for a subtask to have tardiness $k+1$, $kM+1$ subtasks simultaneously miss their deadline at time $t_d$. Hence, $LAG(\tau, t_d) \geq kM+1$, implying the existence of time $t$ as follows: $t < t_d$ and $LAG(\tau, t) < kM+1$ and $LAG(\tau, t+1) \geq kM+1$.

The rest of the proof is the same except for Lemma 5.12, which is modified as follows.

**Lemma 5.15.** *For $U \in A$, $lag(U, t+1) < (k+1) \times wt(U)$.*

*Proof.* Let $U_j$ be the subtask of $U$ scheduled in slot $t$. Since $t < t_d$, $U_j$ has a tardiness of at most $k$. Since $U_j$ completes at time $t+1$, we have $d(U_j) \geq t+1-k$. Hence, by (4.4),

$r(U_{j+1}) \geq t-k$. Since $U_j$ is scheduled in slot $t$ in $S$, it follows that the excess share in the ideal schedule over $[0, t+1)$ is at most $f(U_{j+1}, t-k)$ plus the share over $[t-k+1, t+1)$. By (F1), the second term is at most $k \times wt(U)$. Further, as in Lemma 5.12, we can show that $f(U_{j+1}, t-k) < wt(T)$. Therefore, $LAG(U, t+1) < (k+1) \times wt(U)$. ■

In other words, the lag of a task $U$ in $A$ may be up to $(k+1)$ times the weight of $U$. By Lemmas 5.10, 5.11, and 5.15, $LAG(\tau, t+1) < \sum_{U \in A}(k+1) \times wt(U)$. By (Mk), the right-hand-side of this inequality is at most $M+1$. Thus, $LAG(\tau, t+1) < M+1$; a contradiction. Thus, we have the following result.

**Theorem 5.8.** *Every feasible IS task system satisfying* (Mk) *has a tardiness of at most $k$ under* EPDF *on $M$ processors.*

Condition (Mk) provides us an individual weight restriction of $\dfrac{kM+1}{(k+1)(M-1)}$.

**Corollary 5.8.1.** EPDF *guarantees a tardiness of at most $k$ on $M$ processors for any dynamic IS task system satisfying* (J1) *and* (L1) *if the weight of each task is at most* $\dfrac{kM+1}{(k+1)(M-1)}$.

As done in Lemma 5.13, we can improve (Mk) as follows. (The proof is similar because Lemma 5.14 holds.)

**Lemma 5.16.** *Let $\tau$ be a task system that satisfies* (J1) *and* (L1) *and let $w_1, w_2, \ldots, w_{M-1}$ denote the weights (in non-increasing order) of the $M-1$ heaviest tasks in $\tau$. If $w_{M-1} + (k+1) \times \sum_{i=1}^{M-2} w_i \leq kM+1$, then* EPDF *guarantees a tardiness of at most one for $\tau$.*

This allows us to improve the individual weight restriction to $\dfrac{kM+1}{(k+1)(M-2)+1}$; if every task has weight at most $\dfrac{kM+1}{(k+1)(M-2)+1}$, then $w_{M-1} + (k+1) \times \sum_{i=1}^{M-2} w_i$

$$
\begin{aligned}
&\leq \frac{kM+1}{(k+1)(M-2)+1} + \frac{(k+1)(M-2)(kM+1)}{(k+1)(M-2)+1} \\
&= (kM+1) \times \left(\frac{1+(k+1)(M-2)}{(k+1)(M-2)+1}\right) \\
&= kM+1
\end{aligned}
$$

Thus, we have the following result.

**Corollary 5.8.2.** EPDF *guarantees a tardiness of at most one on $M$ processors for any dynamic IS task system satisfying* (J1) *and* (L1) *if the weight of each task is at most* $\dfrac{kM+1}{(k+1)(M-2)+1}$.

### 5.2.3 Experimental Evaluation

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

The following procedure was followed in every run of the simulation. First, the number of processors was chosen as a random number between 1 and 32. Then, task weights were generated randomly in the interval $(0, 1]$. (No weight restrictions were applied, *i.e.*, weights were allowed to be as large as one.) The weight of the last task was chosen so that the total weight equaled the number of processors. For each generated task set, we constructed an EPDF schedule over the time interval $[0, 10L)$, where $L$ is the least common multiple of all task periods. We repeated this procedure approximately 195,000 times. Thus, the number of data points obtained per processor in each graph is approximately 6,000.

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

The graph in Figure 5.6 plots the percentage of task sets with tardiness greater than zero versus the number of processors. (Recall that a task set has tardiness greater than zero even if just a *single* job misses its deadline.) For example, EPDF misses at least one deadline for approximately 19% of the generated task sets on five processors. (No task set misses a deadline for systems of one or two processors because of the optimality of EPDF for such systems (refer to Corollary 5.1.2.) The percentage of task sets with non-zero tardiness initially increases as the number of processors increases and then steadies. The maximum is around 25%. Though this value may seem high, as the remaining graphs show, the fraction of deadlines that were missed tended to be extremely low.

In Figure 5.7, the average percentage of deadlines missed is shown versus the number of processors. Inset (a) shows the average over all generated task sets, while inset (b) shows the average over task sets that have at least one deadline miss. Both the percentage of job deadlines missed and the percentage of subtask deadlines missed are

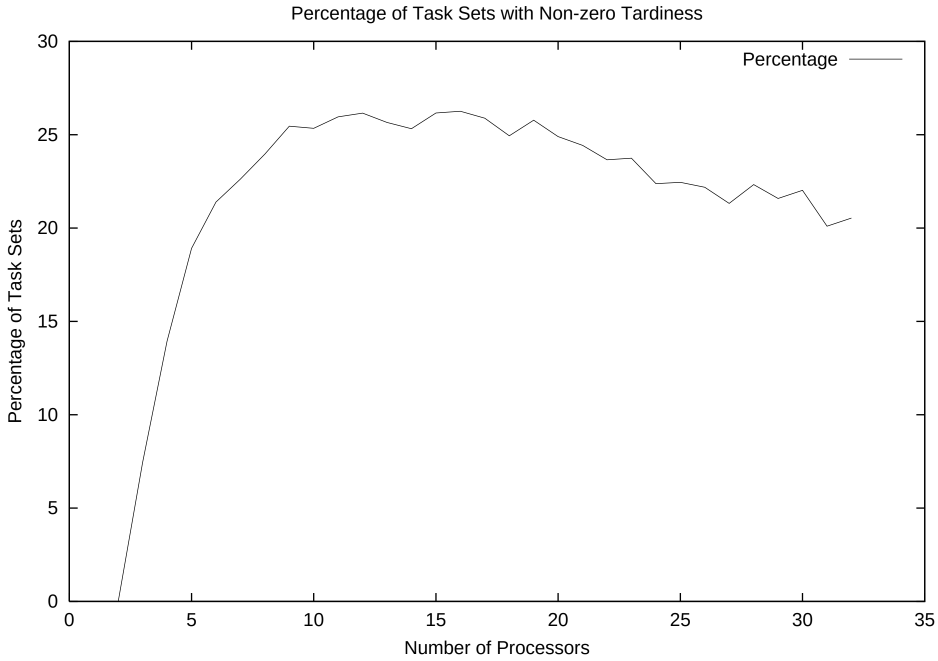


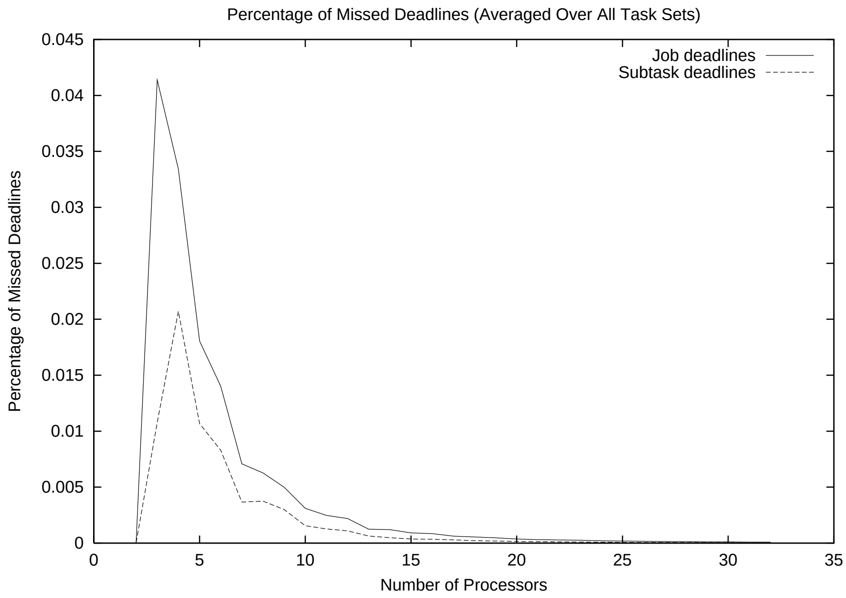

Figure 5.6: Percentage of task sets with non-zero tardiness versus the number of processors.

plotted. As can be seen from these graphs, deadline misses are quite rare.

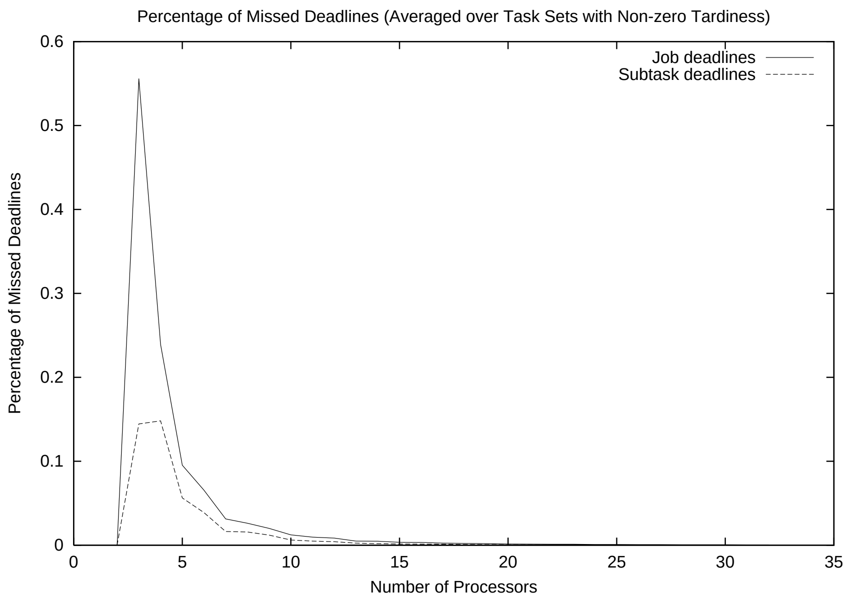
Note that the percentage of job deadlines missed is more than the percentage of subtask deadlines missed. There are two reasons for this. First, the total number of subtask deadlines is much more than the total number of job deadlines. Second, subtask deadline misses within a job have a tendency to cascade, causing the later subtasks within a job (and hence the job itself) to have a higher likelihood of missing their deadlines.

Surprisingly, the largest average of job deadlines missed in Figure 5.7 is obtained for three processors: 0.04% in inset (a) and 0.55% in inset (b). It is also surprising that the percentage of misses decreases as the number of processors increases. Recall from Section 5.1.2 that EPDF is optimal on $M$ processors if the weight of each task is at most $\frac{1}{M-1}$. (Also recall that this condition is fairly tight.) As $M$ increases, $\frac{1}{M-1}$ decreases and hence, intuitively, the chance of a deadline miss should also increase.

However, as $M$ increases, more tasks need to be generated to fully utilize the system. Because our samples are generated randomly, the probability of having low-weight tasks in the system also increases. This in turn drives the number of tasks (and hence, the total number of deadlines) up. Thus, though the number of missed deadlines may




**(a)**




**(b)**

Figure 5.7: Solid (dotted) lines denote the percentage of job (subtask) deadlines missed. **(a)** Percentage of deadlines missed averaged over all task sets. **(b)** Percentage of deadlines missed averaged over task sets with non-zero tardiness. (99% confidence intervals were computed for each graph but are not shown because the relative error associated with each point is very small — less than 0.2% of the reported value.)

increase, the percentage of deadline misses decreases. Analogously, for fewer processors, deadline misses are more common when most tasks have a large weight. In this case, the number of tasks is small and therefore, the percentage of deadlines missed tends to be higher.

## 5.3 Summary

In this section, we studied the scheduling of hard and soft real-time task systems under the EPDF scheduling algorithm. Though EPDF is not an optimal scheduling algorithm, it is simpler and more efficient than $PD^2$. Hence, EPDF is preferable for use in soft real-time systems; under most cases, EPDF guarantees that no subtask (and hence no job) misses its deadline by more than one time unit. Further, even for hard real-time systems, EPDF is preferable under certain circumstances (refer to Theorems 5.1 and 5.6).

# Chapter 6

# Scheduling of Aperiodic Tasks*

Until now, we have only considered the scheduling of recurrent real-time tasks. In addition to such tasks, a system may consist of non-recurring (and maybe non-real-time) tasks. Examples include service routines that are invoked infrequently, including those used for handling and processing interrupts. Such tasks can be modeled using the aperiodic task model.

An *aperiodic task* consists of a single job that arrives at an arbitrary time. An aperiodic task may be either *hard* or *soft*; a hard aperiodic task has a deadline, while a soft aperiodic task does not. Whether an arriving hard aperiodic task's deadline can be guaranteed is a function of the current system workload; thus, such tasks must be subject to an admission-control test. Usually, the goal is to admit as many hard aperiodic tasks as possible and to minimize the response times of soft aperiodic tasks, without causing recurrent tasks to miss their deadlines. Several researchers have addressed this problem for uniprocessor systems [CC89, DLS97, GB95, LRT92, LSS87, SSL89, SB96].

In this chapter, we present several schemes for multiplexing aperiodic tasks and real-time GIS tasks. (Recall that in the GIS task model, the notion of an IS task is generalized by allowing a task to skip subtasks.) Our approaches are server-based, *i.e.*, a recurrent task is used as a server to schedule aperiodic tasks whenever it is scheduled. This is the one of the most common approaches for scheduling aperiodic tasks on uniprocessors.

An aperiodic task is characterized by two integer parameters: its release time (or

---

*The results presented in this chapter have been published in the following paper.

[SHA02] A. Srinivasan, P. Holman, and J. Anderson. Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 19–28, June 2002.

arrival time) and its worst-case execution time. A task's release time is usually not known before its actual arrival. Hard aperiodic tasks have an additional relative deadline parameter; for such tasks, the execution time and deadline must be known upon arrival. For simplicity, we assume that all aperiodic tasks in a system are either hard or soft, *i.e.*, both kinds are not simultaneously present.

On a uniprocessor, aperiodic tasks can be serviced by a single server task. On a multiprocessor, if the available processor capacity exceeds one, then it cannot be completely utilized by a single server unless it is scheduled in parallel with itself. (Even if the spare capacity is at most one, the available parallelism might be more fully exploited by using multiple servers.)

In this chapter, we start by discussing the single server case in Section 6.1, and then consider multiple servers in Section 6.2. Later, in Section 6.3, we describe the results of experiments that we conducted to compare our approaches to background scheduling.

## 6.1 The Single Server Case

In this section, we assume that the total utilization of the system including the aperiodic server is $M$, the number of processors. In other words, we assume that the weight of the aperiodic server is $M - \sum_{T \in \tau} wt(T)$ and that this weight is at most one. Note that, with a single server, it is sufficient to have a single global queue for the aperiodic tasks.

Whenever the aperiodic server is scheduled, it selects an aperiodic task for execution if the queue is not empty. Since the earliest-deadline-first (EDF) algorithm is optimal on a single processor, we assume that the hard aperiodic tasks are scheduled using EDF. On the other hand, we do not assume anything about algorithm used for scheduling soft aperiodic tasks. The approaches that we present in this chapter can be used with any scheduling policy such as first-come first-serve (FCFS) and shortest job first. (However, in our experiments, we use FCFS for scheduling; under FCFS, it is not necessary to know the task execution requirements for scheduling.)

We assume that the real-time GIS tasks in the system are scheduled in a Pfair manner, *i.e.*, they are not early-released. In other words, for each subtask $T_i$, $e(T_i) = r(T_i)$. It is not necessary that the real-time tasks be scheduled in a Pfair manner; however, early-releasing them would probably increase aperiodic response times, because allowing subtasks of the real-time tasks to execute early can delay the execution of the aperiodic server.

In the following subsection, we describe a generic admission-control test for hard aperiodic tasks. In later subsections, we describe the two kinds of servers considered in this chapter: Pfair servers and ERfair servers. We also discuss how to evaluate worst-case response times for aperiodic tasks scheduled using these servers.

### 6.1.1 Admission-control Test for Hard Aperiodic Tasks

We assume a generic *response-time function* $R\colon \mathcal{N} \mapsto \mathcal{N}$, *i.e.*, $R(e)$ gives the worst-case response time for $e$ units of execution (by any number of aperiodic tasks).[1] In other words, starting at time $t$, the server is guaranteed to execute for $e$ time units by time $t + R(e)$. The actual evaluation of the response-time function depends on the server implementation and will be considered later.

Suppose that, at time $t$, $k$ aperiodic tasks are released. Let $A$ denote the set of these tasks and let $d'_i, 1 \leq i \leq k$, be their deadlines. Assume that $d'_i \leq d'_{i+1}$ for $1 \leq i < k$. Merge this list of deadlines with the list of deadlines of aperiodic tasks admitted previously. Let this merged list be $d_1, d_2, \ldots, d_n$, where $n$ is the total number of aperiodic tasks including the newly-released ones, and $d_i \leq d_{i+1}$ for $1 \leq i < n$. Let $T^{(i)}$ be the task with deadline $d_i$. Let $e_i$ be the *remaining* execution time for $T^{(i)}$.

Figure 6.1 shows a simple admission-control algorithm that can be implemented in $O(k \log k + (n+k)f)$ time (explained later), where $f$ is the time complexity of the response-time function. The idea behind the algorithm is as follows. First, note that all tasks admitted prior to the current time have been guaranteed to meet their deadlines. Thus, in line 1, the set $C$ is initialized to include all previously-accepted tasks with a deadline at most $d'_1$. In lines 5–22, the remaining aperiodic tasks are considered in deadline order.

At each step, we do the following. Let the current task being considered be $T^{(i)}$. Tentatively admit this task and then check whether its deadline can be guaranteed (lines 6–9). If not, then there are two cases to consider. If $T^{(i)} \in A$ (refer to line 9), then $T$ is rejected. Otherwise, we reject some other task in $A$ that has already been added to $C$.[2] (Such a task exists because $T^{(i)} \notin A$ is guaranteed to meet its deadline if no new tasks are admitted.) We repeat this rejection process until the deadline of $T^{(i)}$ can be guaranteed.

[1]It is not necessary for $e$ to be an integer. However, this assumption simplifies the analysis; our approach can be easily adapted to non-integral execution times.

[2]Note that, in line 15, the tasks are rejected in order of non-increasing execution times. However, this is not necessary, and any other parameter indicating the "importance" of the newly-admitted tasks may be used.

1: $C := \{T^{(i)} \mid d_i \leq d'_1$ and $T^{(i)} \notin A\}$;
2: $E := \sum_{T^{(i)} \in C} e_i$;
3: $WCR := R(E)$;
4: Let $T^{(j)}$ be the task with the largest index in $C$;
5: **for** $i = j + 1$ to $n$ **do**
6: $\quad C := C \cup \{T^{(i)}\}$;
7: $\quad E := E + e_i$;
8: $\quad WCR := R(E)$;
9: $\quad$ **if** $(t + WCR > d_i) \wedge (T^{(i)} \in A)$ **then**
10: $\quad\quad$ **reject** task $T^{(i)}$;
11: $\quad\quad C := C - \{T^{(i)}\}$;
12: $\quad\quad E := E - e_i$
13: $\quad$ **else**
14: $\quad\quad$ **while** $(t + WCR > d_i)$ **do**
15: $\quad\quad\quad$ Let $e_r = \mathbf{max}\{e_l \mid T^{(l)} \in C \cap A\}$;
16: $\quad\quad\quad$ **reject** task $T^{(r)}$;
17: $\quad\quad\quad C := C - \{T^{(r)}\}$;
18: $\quad\quad\quad E := E - e_r$;
19: $\quad\quad\quad WCR := R(E)$
20: $\quad\quad$ **od**
21: $\quad$ **fi**
22: **od**

Figure 6.1: A generic admission-control procedure. The aperiodic tasks are considered in order of non-decreasing deadlines. The set $C$ represents the aperiodic tasks that have been tentatively accepted; some of the tasks in $C \cap A$ might be rejected at some later point in the execution of this algorithm. $E$ represents the total remaining execution time of the tasks in $C$. $WCR$ represents the worst-case response time of the tasks in $C$. Task $T^{(r)}$ represents the task chosen for rejection and $e_r$ is its execution time.

**Correctness.** At the end of the procedure, the set $C$ represents all the admitted aperiodic tasks in the system. Note that, for each task in $C$, the worst-case completion time $(t + WCR)$ is at most its deadline; otherwise, some task in $C \cap A$ is removed until this holds (refer to lines 9, 11, 14, and 17). Thus, every task in $C$ is guaranteed to meet its deadline. Note also that a task that has been previously admitted is never rejected.

**Time complexity.** We can sort the newly-released $k$ tasks in order of non-decreasing deadlines in $O(k \log k)$ time. Merging this list with the list of already-admitted tasks takes $O(n)$ time. We now show that the response time function is invoked at most $O(n + k)$ times. Note that each task is added at most once to $C$ ($O(n)$ time), and each newly-released task is removed at once from $C$ ($O(k)$ time). This implies that the set

$C$ changes at most $O(n+k)$ times. Since the response time function is invoked only if there is a change in $C$, it is invoked at most $O(n+k)$ times. The **max** calculation on line 15 can be done in $O(1)$ time by maintaining an additional ordered list of the newly-released tasks in $C$. Thus, the procedure runs in $O(k \log k + (n+k)f)$ time.

### 6.1.2 Pfair Servers

In this approach, the aperiodic server is scheduled in a Pfair manner, *i.e.*, each subtask of the server is scheduled in its PF-window. Whenever the server is scheduled, it schedules some unfinished aperiodic task. If no such task exists, then the server has three options, which are described below. Let $t$ be such a time, let $S$ denote the server task, and let $S_i$ denote its eligible subtask at time $t$. Figure 6.2 illustrates the difference between the three variants using the following task set scheduled on two processors: a set $Y$ consisting of four tasks of weight $1/4$, and a set $Z$ consisting of 22 tasks of weight $1/32$. The server task $S$ has weight $2 - 4 \times 1/4 - 22 \times 1/32 = 5/16$. An aperiodic task $A$ is released at time 2 with an execution requirement of 2. Thus, the server has no task to schedule until time 2. The three options are as follows.

- *Idle* the processor. This is the simplest and most efficient scheme, but processor time is wasted.

  This variant is illustrated in Figure 6.2(a). At time 0, subtask $S_1$ has the highest priority according to $PD^2$. Since the queue is empty at time 0, $S$ just idles the processor.

- *Drop* $S_i$ (*i.e.*, consider it absent) and schedule some other eligible subtask. This server is implemented as a GIS task. This variant does not waste processor time.

  Consider the example in Figure 6.2(b). Note that there are two tasks scheduled in slot 0 as opposed to one in the idling variant case. However, $A$'s completion time is the same as for the idling server.

- *Stall* the release of $S_i$ until the next slot and declare it ineligible at $t$. This server is implemented as an IS task. More precisely, if the stalling server is scheduled when the aperiodic task queue is empty, it withdraws its current subtask $S_i$ (*i.e.*, $S_i$ is viewed as having not yet been released) and changes the release time of $S_i$ to be the next slot. By delaying $S_i$'s release, its deadline is also delayed and hence its priority relative to other subtasks drops. This method defers the release of

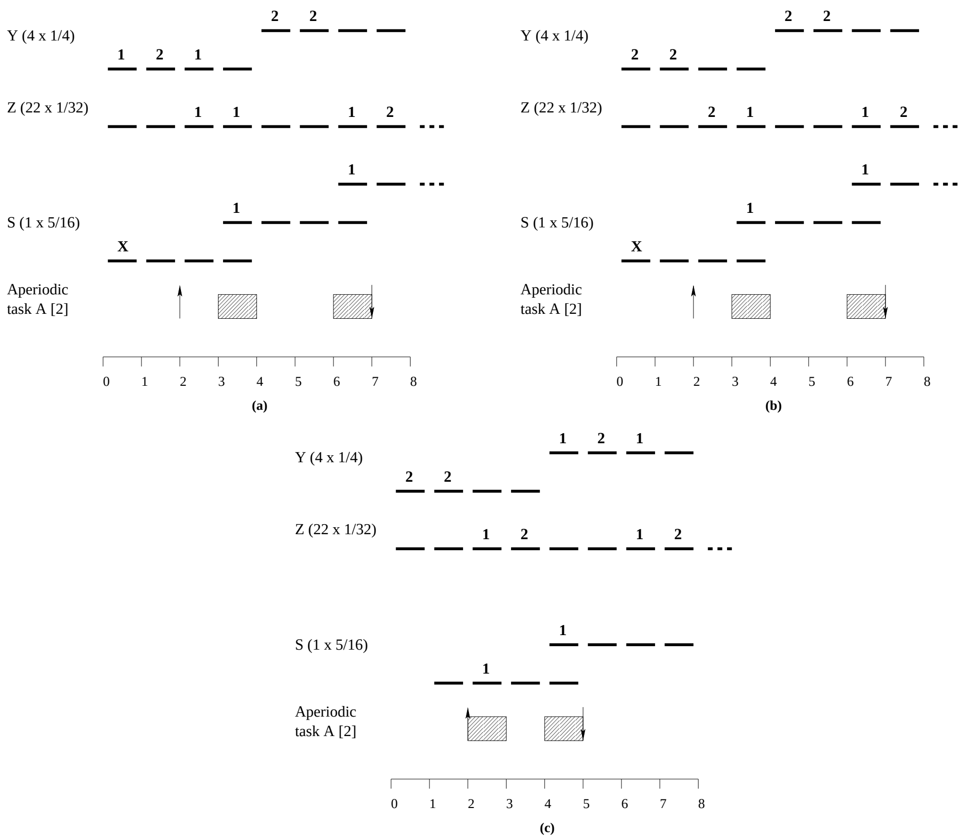



Figure 6.2: A partial schedule for a task set is shown under the different Pfair server variants. An 'X' in slot $t$ refers to the situation where the Pfair server $S$ is scheduled but the aperiodic task queue is empty. The aperiodic task is shown on a different line. The value in brackets indicates its execution time. The up-arrow and down-arrow denote its release time and finish time, respectively. **(a)** The idling variant. **(b)** The dropping variant. **(c)** The stalling variant.

$S_i$ to the last moment possible (but, as our example shows, $S_i$ may be released *before* the next aperiodic task arrives). Intuitively, this scheme should perform better than the previous two schemes and our experiments confirm this.

Figure 6.2(c) depicts an example execution of a stalling server. At time 0, $S$ postpones the release of its next subtask until the next time slot, *i.e.*, slot 1. However, at time 1, there are two subtasks with priority higher than $S$'s subtask; hence, $S$'s subtask is not chosen by PD$^2$ and its release is not further postponed. In the next time slot, $S$ is scheduled and it schedules $A$ in that slot. Note that the response time is much better in this variant.

Note that none of these schemes requires that execution times be known: until a task completes execution, it will remain in the queue and the server will continue to execute it. (Obviously, for hard aperiodic tasks, execution times *are* needed for admission control.) For similar reasons, non-integral execution times pose no problems. When the server finishes executing a task, it selects the next task for execution. It does not really matter whether this switch happens at the end of a quantum or in the middle of a quantum.

**Calculation of the response-time function.** We now present a constant-time procedure for evaluating $R(e)$, *i.e.*, the worst-case response time for $e$ time units of execution when using a single Pfair server.

Recall that a subtask $T_i$ is active at time $t$ if $t \leq d(T_i)$, $T_i$ has not been scheduled yet, and $T_{i-1}$ (if it exists) has already been scheduled. Note that since $T_{i-1}$ has been scheduled, the following is true.

$$i > 1 \Rightarrow r(T_{i-1}) < t \tag{6.1}$$

Suppose the response-time function $R$ is invoked at time $t$. Let $S$ denote the server task, and let $S_i$ be the active subtask of $S$ at time $t$. For example, in Figure 6.2, $t$ is 2 and $i$ is 2, 2, and 1, respectively, for the idling, dropping, and stalling variants. (Recall that "time 2" is the beginning of slot 2; thus, $A$ potentially could be scheduled in slot 2.) Now, by the optimality of the PD$^2$ algorithm, task $S$ is guaranteed to receive $e$ units of execution by the deadline of subtask $S_{i+e-1}$, *i.e.*, $d(S_{i+e-1})$. Therefore, $R(e) \leq d(S_{i+e-1}) - t$. Thus, we have the following.

**Claim 6.1.** *$R(e) \leq d(S_{i+e-1}) - t$, where $t$ at which the function $R$ is invoked, and $S_i$ is the subtask of $S$ active at time $t$.*

The expression $d(S_{i+e-1}) - t$ can be evaluated in $O(1)$ time by maintaining $\theta(S_i)$ and using Equation (4.2). Intuitively, the response time varies inversely with the weight of the server; the larger the weight, the smaller will be the value of $d(S_{i+e-1})$. The following claim confirms this intuition by giving a bound that is independent of the current subtask and time.

**Claim 6.2.** $R(e) \leq \left\lceil \dfrac{e+1}{wt(S)} \right\rceil$.

*Proof.* Let $t$ be the time at which $R$ is invoked and $S_i$ be the subtask of $S$ active at time $t$. We now show that $d(S_{i+e-1})-t \leq \left\lceil \frac{e+1}{wt(S)} \right\rceil - 1$; the required result then follows from Claim 6.1.

Note that $\theta(S_{i+e-1}) = \theta(S_i)$, because at least $e$ units of execution are requested at time $t$. Therefore, by (4.2), we have the following.

$$d(S_{i+e-1}) = \theta(S_i) + \left\lceil \frac{i+e-1}{wt(S)} \right\rceil. \tag{6.2}$$

If $i = 1$, then $\theta(S_1) \leq t$. (Note that it can be less than $t$ for the stalling server.) Therefore, by (6.2), $d(S_{i+e-1}) \leq t + \left\lceil \frac{e}{wt(S)} \right\rceil$, which implies that $d(S_{i+e-1}) - t \leq \left\lceil \frac{e}{wt(S)} \right\rceil \leq \left\lceil \frac{e+1}{wt(S)} \right\rceil$.

In the rest of the proof, we assume that $i > 1$. In this case, by (6.1), $t > r(S_{i-1})$. Now there are potentially two possibilities: either $t \geq r(S_i)$ or $r(S_{i-1}) < t < r(S_i)$. Note that the response time in the second case will clearly be greater than that in the first; therefore, we only need to consider the second possibility. In this case, $S_i$ will be released as early as possible and hence, $\theta(S_{i-1}) = \theta(S_i)$. Hence, by (4.1), $r(S_{i-1}) = \theta(S_i) + \left\lfloor \frac{i-2}{wt(S)} \right\rfloor$. This implies that $t > \theta(S_i) + \left\lfloor \frac{i-2}{wt(S)} \right\rfloor$, and hence, by (6.2), $d(S_{i+e-1}) - t < \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i-2}{wt(S)} \right\rfloor$. Therefore, $d(S_{i+e-1}) - t \leq \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i-2}{wt(S)} \right\rfloor - 1$. Now,

$$\begin{aligned}
& \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i-2}{wt(S)} \right\rfloor - 1 \\
= \; & \left\lceil \frac{e+1+i-2}{wt(S)} \right\rceil - \left\lfloor \frac{i-2}{wt(S)} \right\rfloor - 1 \\
\leq \; & \left\lceil \frac{e+1}{wt(S)} \right\rceil + \left\lceil \frac{i-2}{wt(S)} \right\rceil - \left\lfloor \frac{i-2}{wt(S)} \right\rfloor - 1 \quad , \quad \lceil x+y \rceil \leq \lceil x \rceil + \lceil y \rceil \\
\leq \; & \left\lceil \frac{e+1}{wt(S)} \right\rceil \quad , \quad \lceil x \rceil \leq \lfloor x \rfloor + 1
\end{aligned}$$

Thus, $d(S_{i+e-1}) - t \leq \left\lceil \frac{e+1}{wt(S)} \right\rceil$. ■

### 6.1.3 ERfair Servers

In this approach, the aperiodic server is scheduled in an ERfair manner. Since the server subtasks are allowed to execute before their windows, the (short-term) rate at which an ERfair server services the aperiodic tasks can be higher than that of a Pfair server. Hence, aperiodic response times will likely be better when using an ERfair server as compared to a Pfair server. (On the other hand, ERfairness introduces more jitter into the server execution. Therefore, Pfair servers are better for systems in which jitter is a concern.) As before, we can have three different implementations of this server. Intuitively, the stalling scheme for ERfair servers is expected to outperform all the other servers and our experiments confirm this.

Figure 6.3 illustrates the three ERfair variants for the example task set considered in Figure 6.2. Inset (a) illustrates the idling variant: when the server is scheduled in slot 0, it simply idles the processor. Note that because of the ERfair nature of the server, the server is able to schedule task $A$ in slot 2. The behavior of the dropping variant (inset (b)) is similar. In the stalling variant (inset (c)), because the second subtask is early-released, $A$ finishes by time 4 as opposed to time 5 in the stalling variant of the Pfair server (refer to Figure 6.2(c)).

One interesting point about the stalling scheme for the ERfair server is the manner in which stalling and ERfairness complement each other. Note that, when the server is backlogged (*i.e.*, the queue is non-empty), stalling is not used and the ERfair nature of the server gives good response times. On the other hand, when not backlogged, as noted earlier, the stalling scheme provides the best response time among the three schemes.

**Calculation of the response-time function.** The response-time estimate given for Pfair servers (refer to Claim 6.1) can be used to calculate the worst-case completion times for ERfair servers as well; however, this estimate can differ considerably from the actual response time. The following procedure gives us a more accurate value for ERfair servers. As before, let $S$ denote the server and let $t$ be the time when the response-time function is invoked. Also, let $S_i$ be the subtask of $S$ active at $t$ and let $d = d(S_{i+e})$. (Thus, $e+1$ subtasks of $S$ are guaranteed to execute in the interval $[t, d)$.) The response time $R(e+1)$ is calculated in a recursive manner using the value for $R(e)$. Because $R(e)$ represents the worst-case response time for $e$ time units, subtask $S_{i+e-1}$ is guaranteed to complete by time $t + R(e)$. Thus, $S_{i+e}$ is eligible for execution at or after $t + R(e)$.

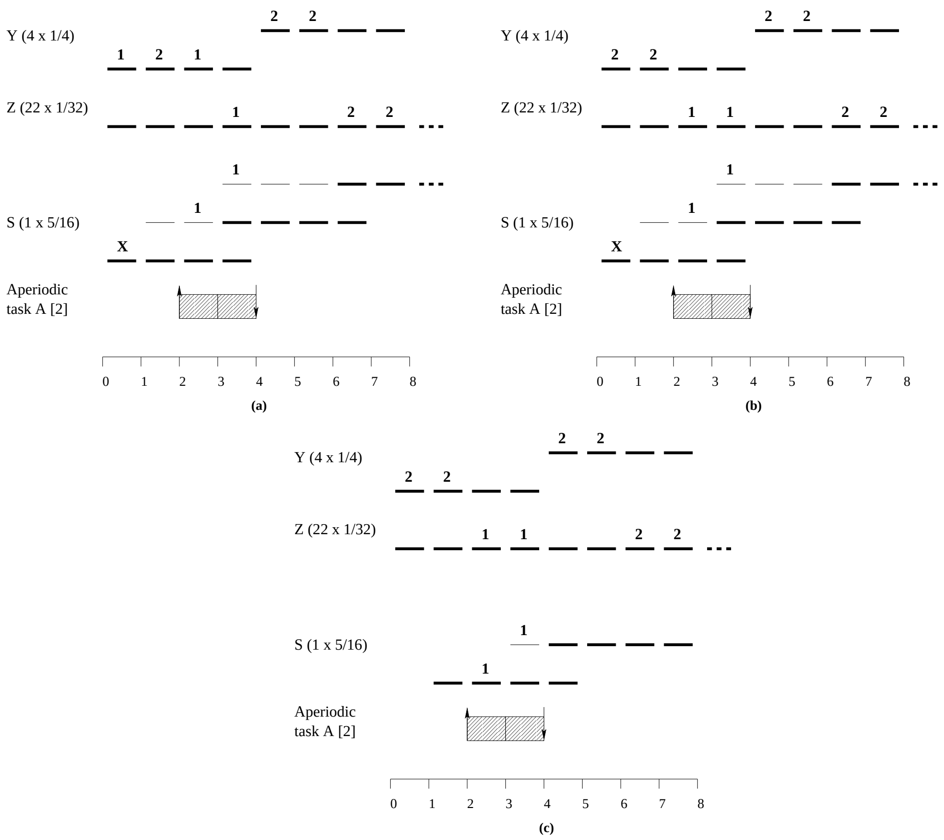



Figure 6.3: A partial schedule for a task set is shown under the different ERfair server variants. **(a)** The idling variant. **(b)** The dropping variant. **(c)** The stalling variant.

Let $H$ denote the set of all subtasks (including those of real-time tasks) that may contend with $S_{i+e}$ in the interval $[t + R(e), d)$ and that have higher priority. These are the only subtasks that can delay the execution of $S_{i+e}$ after time $t + R(e)$. Let $h = |H|$. We now give a bound for $R(e + 1)$ in terms of $R(e)$ and $h$.

**Claim 6.3.** $R(e+1) \leq R(e) + \left\lfloor \frac{h}{M} \right\rfloor + 1.$

*Proof.* Since the number of subtasks in $H$ is $h$, these subtasks can run for at most $\left\lfloor \frac{h}{M} \right\rfloor$ time without leaving a processor idle. (In order to keep all processors busy for an interval of length $\left\lfloor \frac{h}{M} \right\rfloor + 1$, the number of subtasks needed is $(\left\lfloor \frac{h}{M} \right\rfloor + 1) \times M > h$.) Because $S_{i+e}$ is eligible to execute at or after $t+R(e)$, this implies that $S_{i+e}$ is guaranteed to execute in the interval $[t + R(e), t + R(e) + \left\lfloor \frac{h}{M} \right\rfloor + 1)$. ■

The following claim can be used to bound the value of $h$. Let $\theta(U)$ denote the offset of $U$ at time $t$, where $U$ is any task in $\tau$. Let $I(U)$ denote the index of the last subtask of $U$ scheduled before $t$. Further, let $\tau' = \tau - \{S\}$.

**Claim 6.4.** $h \leq \sum_{U \in \tau'} \lfloor (d - \theta(U)) \cdot wt(U) \rfloor - \mathbf{max}(I(U), \lfloor wt(U) \cdot (t + R(e) - \theta(U)) \rfloor + 1)$.

*Proof.* The maximum number of subtasks of $U$ that can be scheduled in $[t + R(e), d)$ is $k - \mathbf{max}(I(U), j)$, where $U_k$ is the latest subtask of $U$ that has a deadline at or before $d$, and $U_j$ is the earliest subtask of $U$ such that $d(U_j) > t + R(e)$.

Then, by (4.2), we have $\theta(U) + \left\lceil \frac{k}{wt(U)} \right\rceil \leq d$. Hence, $\left\lceil \frac{k}{wt(U)} \right\rceil \leq d - \theta(U)$. This implies that $\frac{k}{wt(U)} \leq d - \theta(U)$. Therefore, $k \leq (d - \theta(U)) \cdot wt(U)$.

Similarly, by (4.2), we have $\theta(U) + \left\lceil \frac{j}{wt(U)} \right\rceil > t + R(e)$. Because $t + R(e) - \theta(U)$ is an integer, it follows that $\frac{j}{wt(U)} > t + R(e) - \theta(U)$. This implies that $j > (t + R(e) - \theta(U)) \cdot wt(U)$, and hence, $j \geq \lfloor wt(U) \cdot (t + R(e) - \theta(U)) \rfloor + 1$.

Thus, $k - \mathbf{max}(I(U), j) \leq \lfloor (d - \theta(U)) \cdot wt(U) \rfloor - \mathbf{max}(I(U), \lfloor wt(U) \cdot (t + R(e) - \theta(U)) \rfloor + 1)$. Thus, the required result follows by incorporating the subtasks of all tasks. ■

Given the value of $R(e)$, the expression in Claim 6.4 can be calculated in constant time, and hence, $h$ can be calculated in $O(N)$ time, where $N$ is the number of real-time tasks in the system. To obtain the actual value of $R(e)$, we take the minimum of the values obtained from Claims 6.3 and 6.1. In general, the value obtained by the above procedure is expected to be smaller. As an example, consider the task set in Figure 6.3.

For all variants, when the aperiodic task is released at time 2 and response times calculated using Claim 6.3 are as follows. During the calculation of $R(1)$, $t = 2$ and $h = 1$ because of the first subtask of one task in $Y$. Therefore, $R(1) = \left\lfloor \frac{1}{2} \right\rfloor + 1 = 1$. During the calculation of $R(2)$, $t + R(1) = 3$ and $h = 5$ because of the first subtask of one task in $Y$ and the second subtask of each task in $Y$. Thus, $R(2) = 1 + \left\lfloor \frac{5}{2} \right\rfloor + 1 = 4$. Thus, the aperiodic task is guaranteed to complete before time $t + 4$, *i.e.*, time 6. In fact, it actually completes by time 4, as seen in the figure. The response time estimated using Claim 6.1 is 8 ($d(S_3) - 2 = 10 - 2$), 8 ($d(S_3) - 2$), and 6 ($d(S_2) - 2 = 8 - 2$) for the idling, dropping, and stalling variants, respectively. Thus, Claim 6.3 provides a better estimate.

Although Claim 6.3 provides a more accurate bound for the response time, its time complexity is $O(Ne)$ (where $N$ is the total number of tasks in the system), whereas the bound in Claim 6.1 can be calculated in $O(1)$ time. Thus, there is a trade-off between efficiency and accuracy. A simplistic implementation of the ERfair admission-control test with this new procedure requires $O((n + k)Ne + k \log k)$ time, where $n$ and

$k$ are as defined in Section 6.1.1, and $e$ is the sum of the remaining execution times of all aperiodic tasks. It follows directly from the analysis in Section 6.1.1 because $f = O(Ne)$. Note, however, that when calculating $R(e)$, we also calculate $R(i)$, where $1 \leq i < e$. These values can be stored to avoid recomputing later; thus every invocation of $R$ in the admission-control test takes $O(1)$ time. Thus, the admission-control test completes in $O(Ne + n + k + k \log k) = O(Ne + n + k \log k)$ time.

**Relation to uniprocessor server approaches.** The stalling Pfair server is similar to the constant utilization server (CUS) proposed by Deng *et al.* [DLS97] for scheduling aperiodic tasks on uniprocessors. The CUS server is designed using a sporadic task, which is scheduled together with the real-time tasks using EDF. Like Pfair scheduling, this approach is non-work-conserving: the server is not scheduled if it has used up the execution time for its current period even if a processor is idle and an unfinished aperiodic task exists. The total bandwidth server (TBS) of Spuri and Buttazzo [SB96] eliminates unused processor capacity by allowing the "early-releasing" of jobs. The server releases a new job if the current job completes and unfinished aperiodic tasks exist; the new job's deadline is the same as what it would have been had it not been early-released. Thus, this is very similar to early-releasing of subtasks under ERfair scheduling.

## 6.2 The Multiple Server Case

If spare processor capacity exceeds one, then multiple aperiodic servers may be needed to use all the spare capacity. (Note that the weight of any single server or task is at most one.) In this section, we consider several issues pertinent to systems with multiple servers.

Before continuing, recall Theorem 2.6, which states that in any Pfair schedule, a periodic task $T$ receives at least $\lfloor wt(T) \cdot t \rfloor$ units of processor time in the interval $[0, t)$. In the paragraphs that follow, we discuss some heuristics for determining the number of servers and their weights. We then discuss the relative merits of global versus per-server queues and describe a simple admission-control test (which is an extension of the test given in Section 6.1.1 for a single server). Finally, we present a way to better utilize the available parallelism.

**Number of servers and weight distribution.** Suppose the spare processor capacity is distributed by creating $s$ servers. Let $w_i, 1 \leq i \leq s$, be the weights of these servers, where $0 < w_i \leq 1$. Also, let $l + x = \sum_{i=1}^{s} w_i$, where $l$ is an integer and $0 \leq x < 1$. Thus, $l + x$ is the total spare processor capacity that the servers can consume. For any $t$, $\sum_{i=1}^{s} w_i \cdot t = l \cdot t + x \cdot t$. Now,

$$\sum_{i=1}^{s} \lfloor w_i \cdot t \rfloor \leq \left\lfloor \sum_{i=1}^{s} w_i \cdot t \right\rfloor = \lfloor l \cdot t + x \cdot t \rfloor = l \cdot t + \lfloor x \cdot t \rfloor.$$

Hence, the service guaranteed to $s$ servers by time $t$ is at most that guaranteed to $l+1$ servers of weights $x, 1, 1, \ldots, 1$. Thus, better worst-case response times result using $l+1$ servers with $l$ servers of unit weight. We call this the *greedy* policy.

It is important to note that the analysis given here is only a guideline: actual response times may sometimes be better in the $s$-server case because of the extra parallelism available. However, as our experiments illustrate, average response times are better under the greedy policy.

**Global versus per-server queues.** The strategy here varies depending on whether the aperiodic tasks are hard or soft. In the case of soft aperiodic tasks, where we are more interested in improving average response times, it is better to have a global queue rather than to partition. This is because, with partitioning, a server might be scheduled when its own queue is empty and ready aperiodic tasks are waiting in other queues.

On the other hand, in the case of hard aperiodic tasks, if EDF is used for scheduling, then partitioning might be a better option. This is because global scheduling using EDF can result in poor schedulability [DL78]. One way to alleviate this problem is for the servers to use Pfair scheduling techniques (rather than EDF) when scheduling hard aperiodic tasks. However, in general, the servers themselves may execute at different rates. Thus, the problem is equivalent to a multiprocessor problem in which the processor speeds are not identical. No optimal scheduling algorithm is known for this problem; in fact, Baruah *et al.* [Bar00, Bar01] has shown that the corresponding feasibility problem is intractable.

Assuming that partitioning is used for scheduling hard aperiodic tasks, we can use a bin-packing algorithm such as first-fit (refer to Section 2.1.2), and one of the admission-control tests proposed for the single-server case for task assignment. The overall time complexity will depend on the time complexity of the admission-control test. Suppose there are $k$ servers numbered from 1 to $k$, and suppose that the worst-case response-times are calculated using Claim 6.1. Then, the admission-control test for server takes $O(n_i)$ time for every newly-released aperiodic task, where $n_i$ is the number of tasks assigned to server $i$. Hence, the bin-packing assignment can be done in $O(\sum_{i=1}^{k} n_i) = O(n)$ time per aperiodic task, where $n$ is the total number of aperiodic tasks (assigned to all servers).

**Background servers.** Recall that the real-time tasks are assumed to be GIS tasks, and hence their subtasks may be released late. If such late releases are frequent, then processors may be idle often. Idle slots may also result if a job of a real-time task finishes earlier than its estimated worst-case execution time. Note that if an ERfair aperiodic server has tasks to schedule during a potentially idle time slot, it will do so. However, if a processor is idle, and the number of unfinished aperiodic tasks exceeds the number of eligible servers, then processor time will still be unnecessarily wasted. This is illustrated by the following example.

Consider a two-processor system on which two sets of IS tasks are scheduled: a set $A$ of four tasks of weight $1/4$, and a set $B$ of seven tasks of weight $1/16$. In addition, there is one aperiodic server with weight $2 - 4(1/4) - 7(1/16) = 9/16$. Suppose that four aperiodic tasks with execution times 3, 2, 2, and 1 are released at times 0, 1, 10, and 10, respectively. Figure 6.4 shows the resulting schedule using PD$^2$. From the figure, we can see that the aperiodic tasks finish at times 4, 7, 12, and 13, respectively. Note that the fourth aperiodic task is serviced only after the completion of the third aperiodic task at time 12. Also note that a processor is idle in the interval $[10, 11)$. Thus, the fourth aperiodic task could actually have run in parallel with the third in slot 10.

One way to utilize this extra processor time is to have a set of background servers, one on each processor. Such a server is scheduled if its processor becomes idle while unfinished aperiodic tasks exist. Using background servers results in better utilization of the available parallelism than is otherwise possible. Note that the extra processor time utilized in this way does not need to be charged to the Pfair/ERfair servers.

## 6.3 Performance Studies

We conducted simulations to compare our proposed servers with background scheduling under various conditions. Performance was measured by computing the average aperiodic response time as a function of the utilization of the periodic tasks (*i.e.*, the

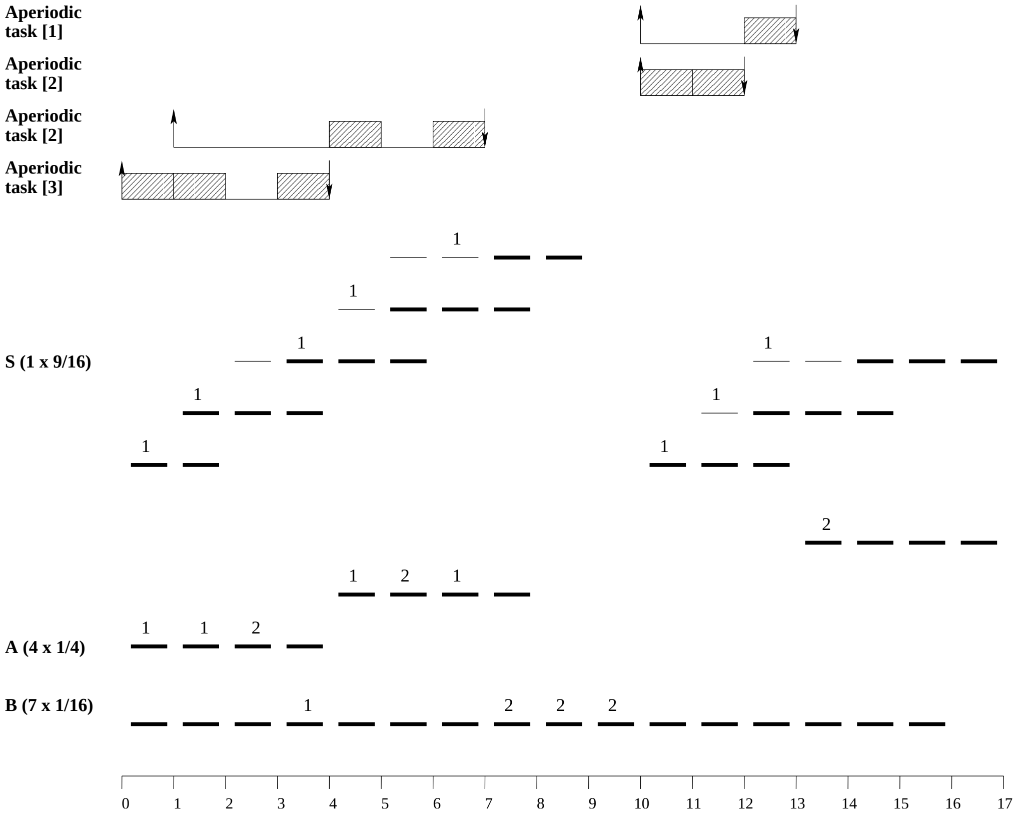



Figure 6.4: A mixed Pfair/ERfair schedule on two processors is depicted. $A$ and $B$ are two sets of real-time GIS tasks scheduled in a Pfair manner and $S$ represents the GIS task used as a stalling ERfair server. Four aperiodic tasks with execution times 3, 2, 2, and 1 are released at times 0, 1, 10, and 10, respectively. A processor is idle over $[10, 12)$, and two unfinished aperiodic tasks exist, but only one of them is scheduled.

spare capacity). Each aperiodic task's response time has been normalized with respect to its execution cost, *e.g.*, a value of 3 indicates a response time that is three times a task's execution cost. Before explaining the results that were obtained, we explain the experimental setup.

There are a number of factors that affect the scheduling of aperiodic tasks on multiprocessors, among them, the number of processors, the total utilization of the periodic tasks, and the distribution of the aperiodic task arrivals. We varied these three parameters to determine their effect on response times. In each experiment, the number of servers and weights was determined using the greedy policy described in Section 6.2. Also, a single global queue was maintained for the aperiodic tasks.

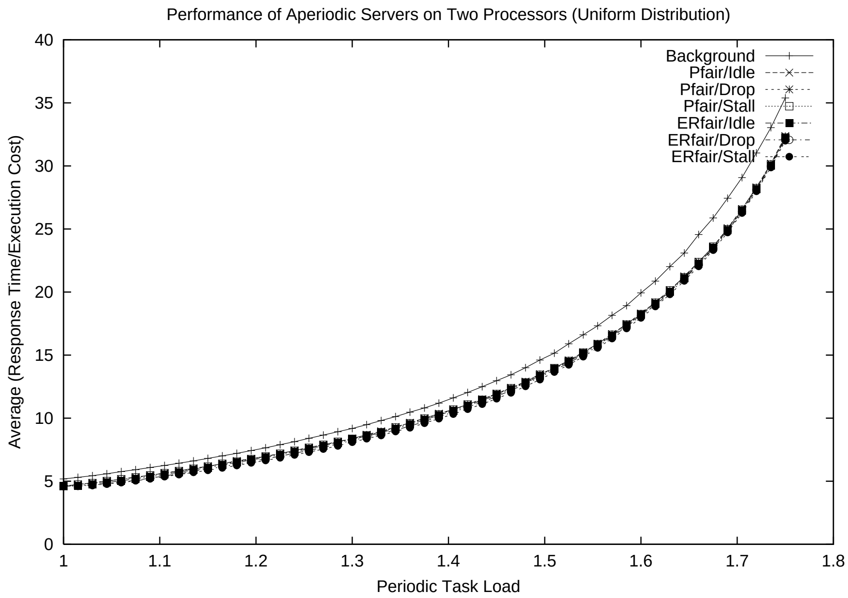
Fifty different periodic task sets having the same total utilization were generated, along with fifty different aperiodic task sets, and simulations were conducted for each combination of these task sets. Thus, each point in each graph is the result of 2,500 simulations. Each line in each graph was obtained by 50 such points uniformly distributed in the interval $[M/2, M - 0.25]$, where $M$ is the number of processors. In other words, the spare capacity was varied from 0.25 to $M/2$ and the effect on the response times was then determined. We conducted these experiments for 2, 4, 8, 16, and 32 processors. We also studied how changing the distribution of the aperiodic arrivals from uniform to bursty affects the performance of the servers.

**Results.** The graphs in Figures 6.5–6.14 illustrate the performance of the six servers described in this chapter.[3] Inset (a) in each figure compares the performance of these servers to background scheduling, and inset (b) demonstrates the performance of these servers relative to an idling Pfair server (which is the most simplistic server, and thus an obvious choice for a baseline measurement).

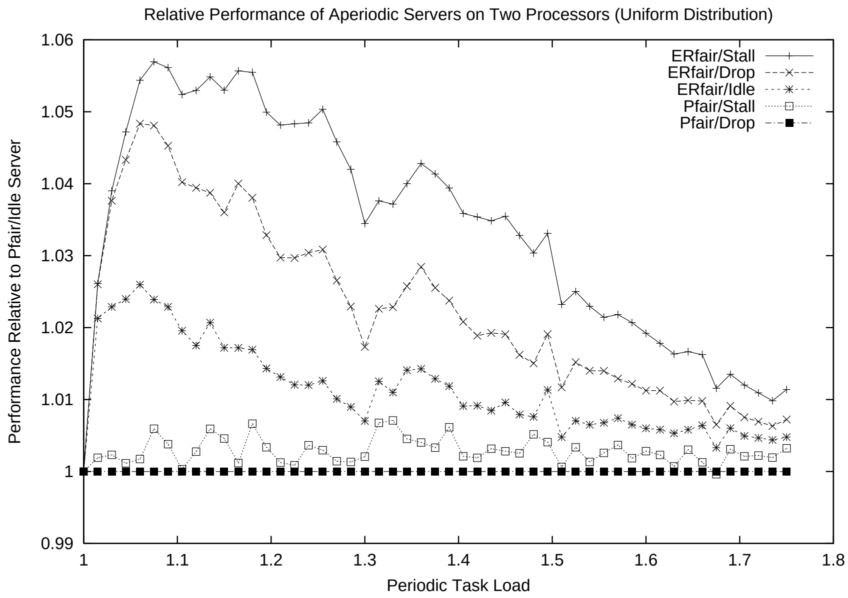
Figures 6.5 and 6.6 illustrate the performance of the servers on two processors. There is one server here, whose weight is varied from 0.25 to 1. In Figure 6.5, the aperiodic task releases are uniformly spaced. As can be easily seen, all the server schemes give better performance than background scheduling; as the periodic task load increases, the improvement (see inset (a)) approaches 10%. As expected, the stalling variant of the ERfair server provides the best performance; as seen in inset (b), its performance is up to 5% better than the idling Pfair server. In Figure 6.6, the aperiodic task releases are bursty and grouped together near time zero. Similar results are seen here, although the improvement over background scheduling is a bit better (approaching about 14%). Note that, in this case, the server is continuously backlogged and hence the differences between the three ERfair variants (see inset (b)) are less pronounced. (Recall that the three variants considered earlier come into play when the servers are idle.) From insets (a) in the both figures, note that as the periodic load increases, the response times also increase (which agrees with intuition, since the utilization of the server decreases).

Figures 6.7, 6.9, 6.11, and 6.13 illustrate the performance of the servers on 4, 8, 16, and 32 processors with uniform aperiodic task arrivals. As seen in insets (a) of these figures, the servers perform even better here compared to background scheduling, approaching 27% in Figure 6.7 and 124% in Figure 6.11. This illustrates the scalability of our approach as the number of processors increases. In fact, for a 32-processor

[3] As we did in the experiments described in earlier chapters, 99% confidence intervals were computed for each graph but are not shown because the relative error associated with each point is very small (less than 1% of the reported value).

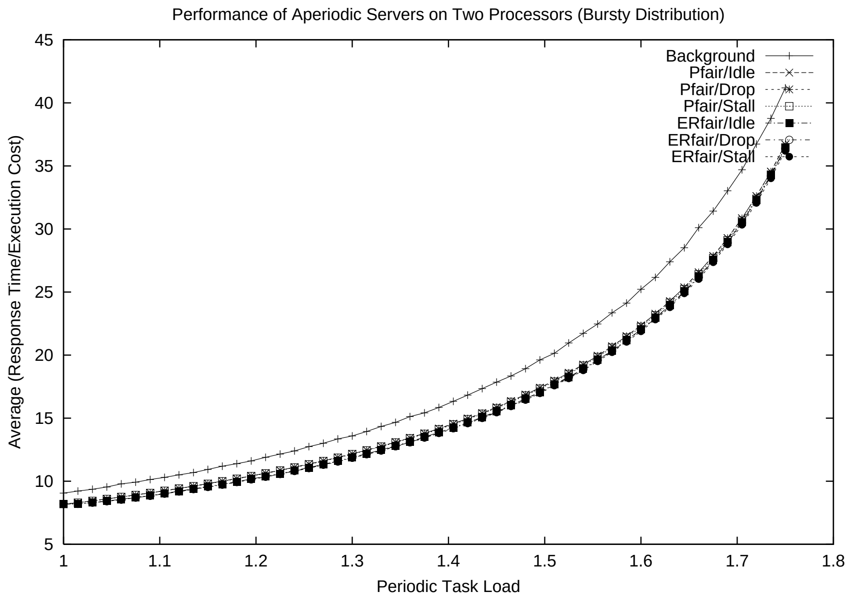



**(a)**

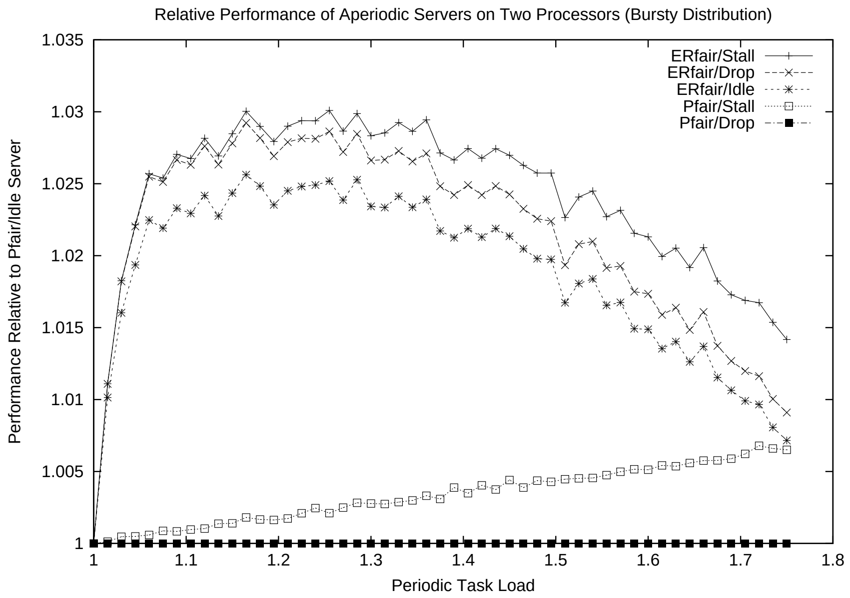



**(b)**

Figure 6.5: Simulation results on two processors with uniform distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

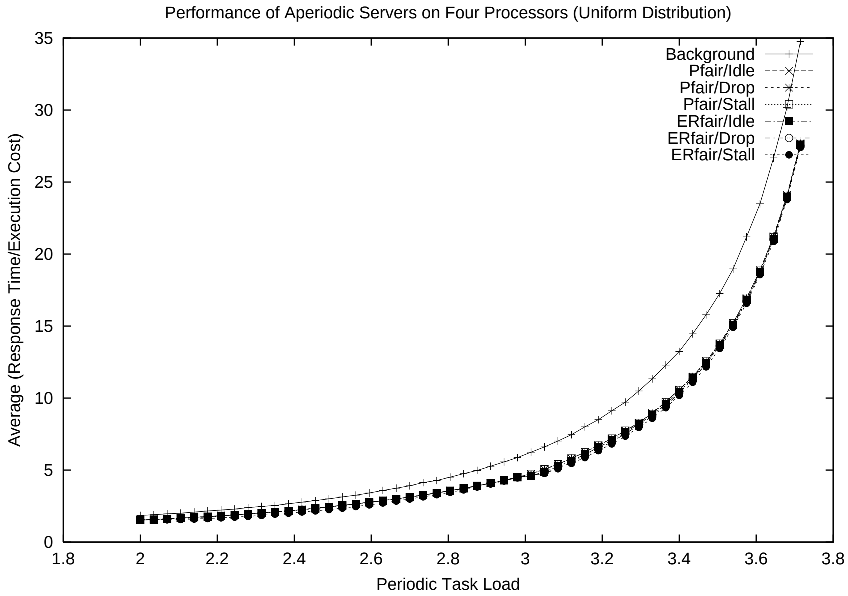



**(a)**

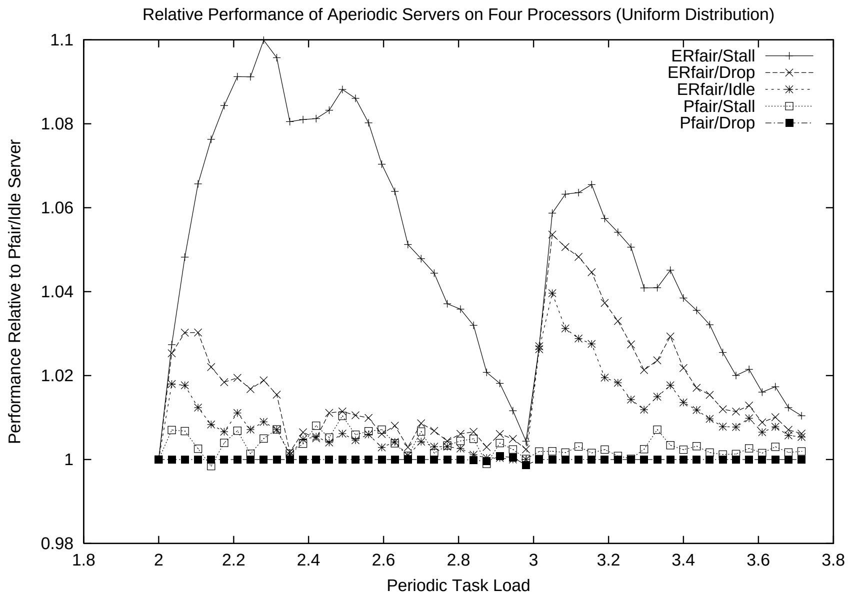



**(b)**

Figure 6.6: Simulation results on two processors with bursty distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

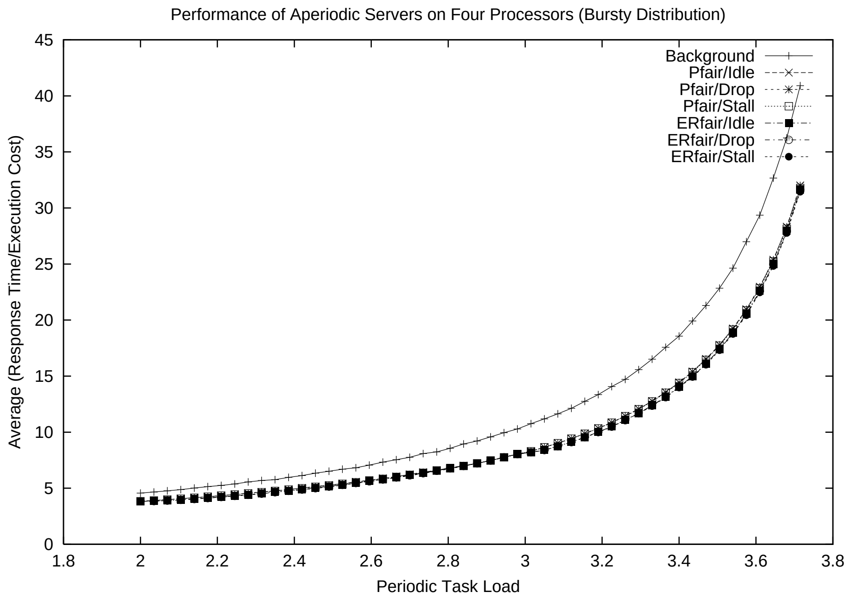



**(a)**

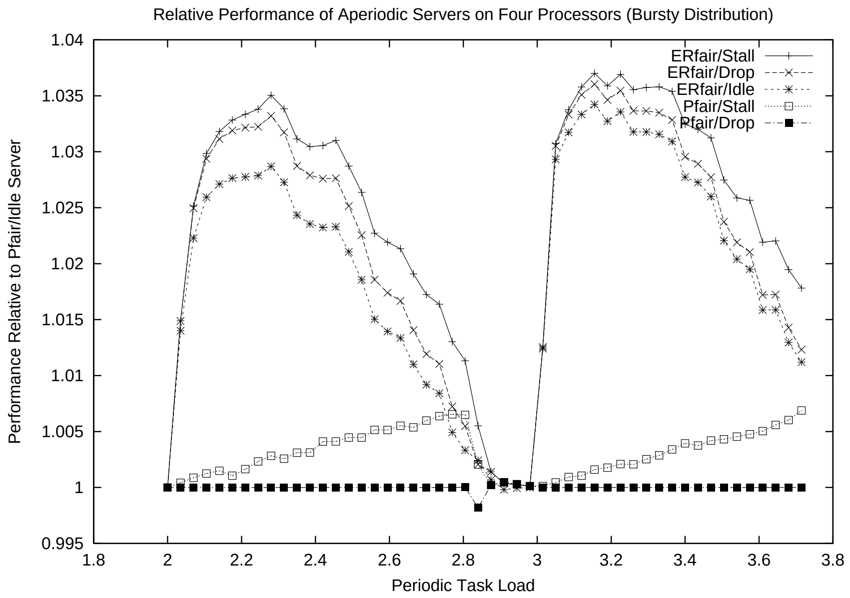



**(b)**

Figure 6.7: Simulation results on four processors with uniform distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

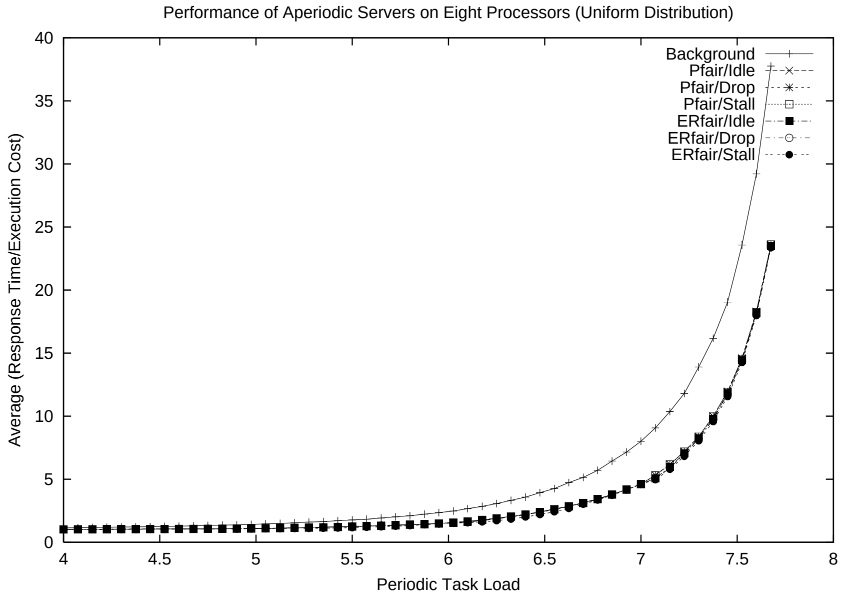



**(a)**

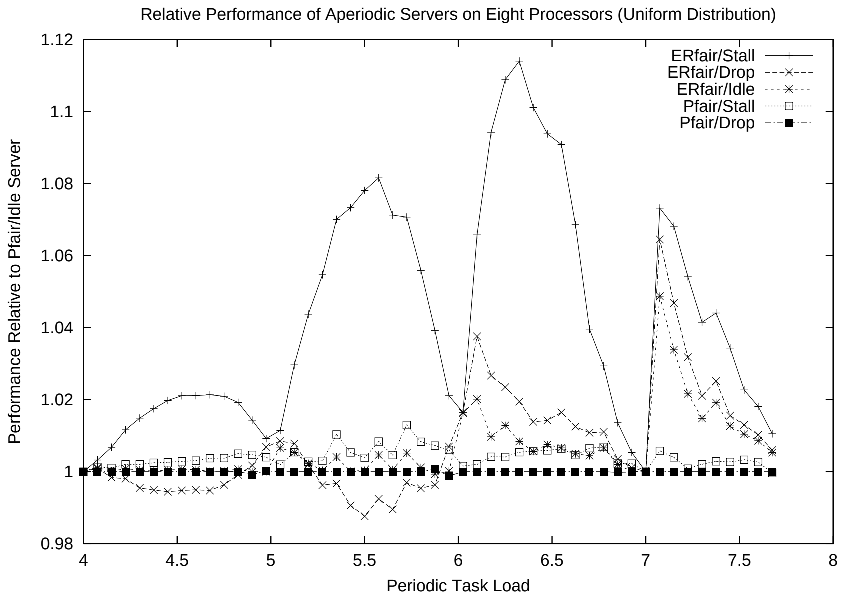



**(b)**

Figure 6.8: Simulation results on four processors with bursty distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

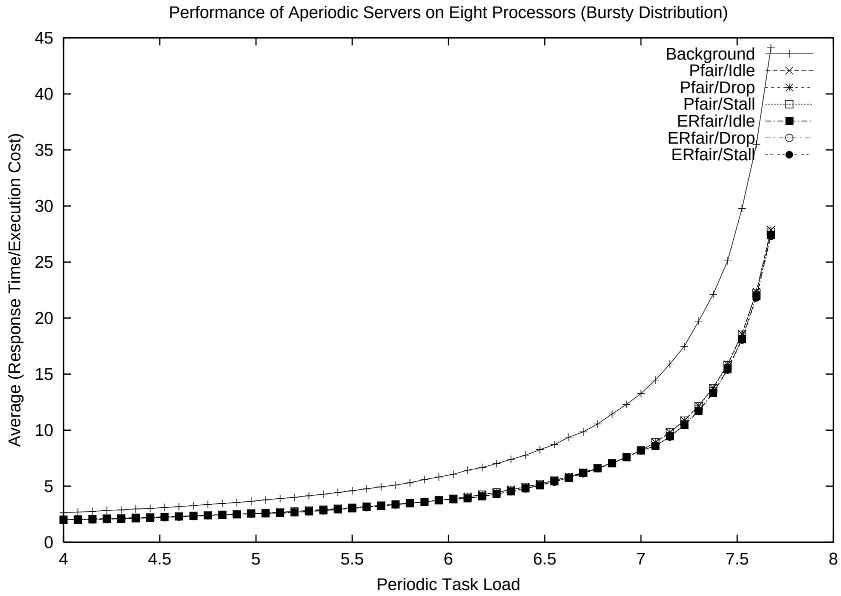



**(a)**

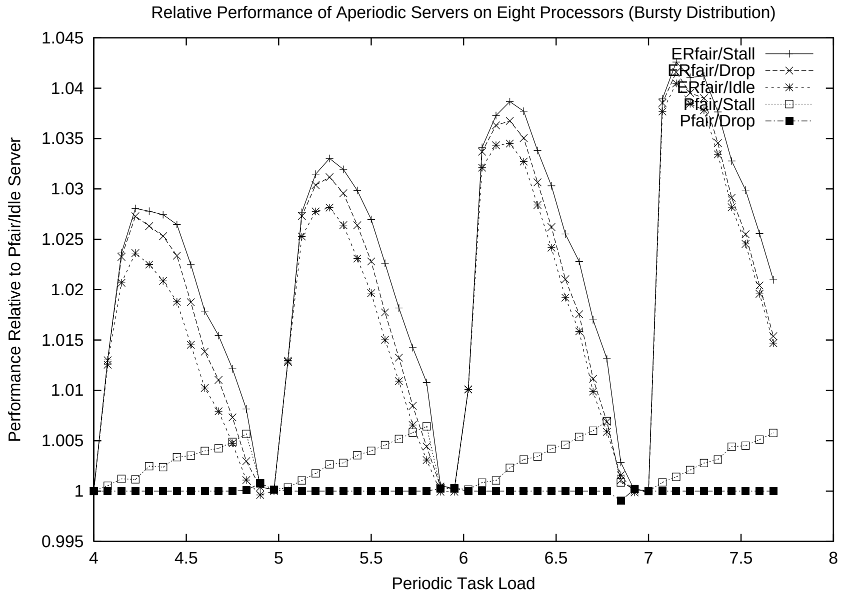



**(b)**

Figure 6.9: Simulation results on eight processors with uniform distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

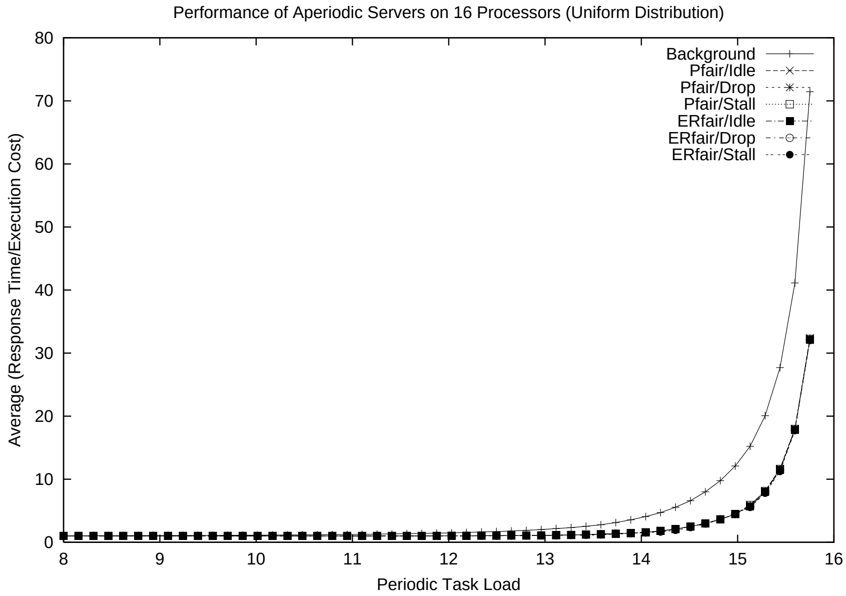



**(a)**

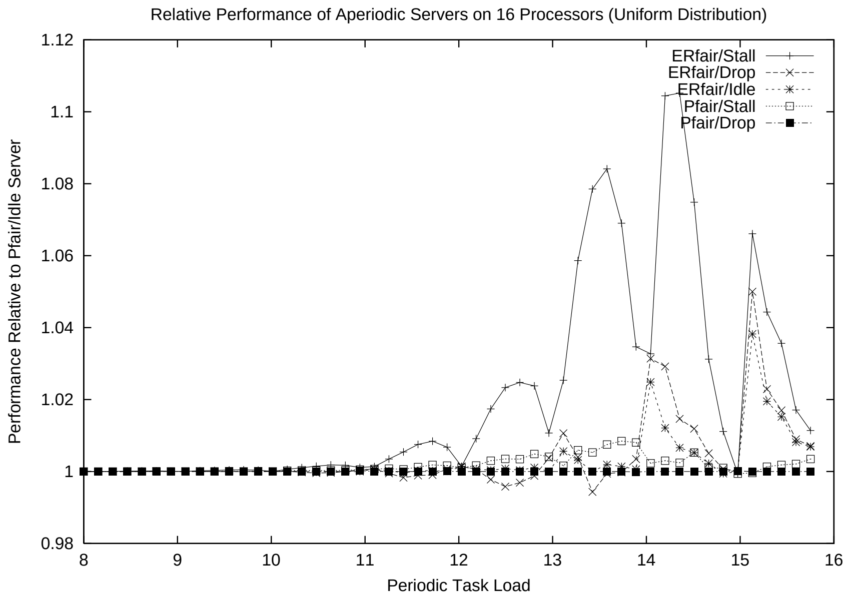



**(b)**

Figure 6.10: Simulation results on eight processors with bursty distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

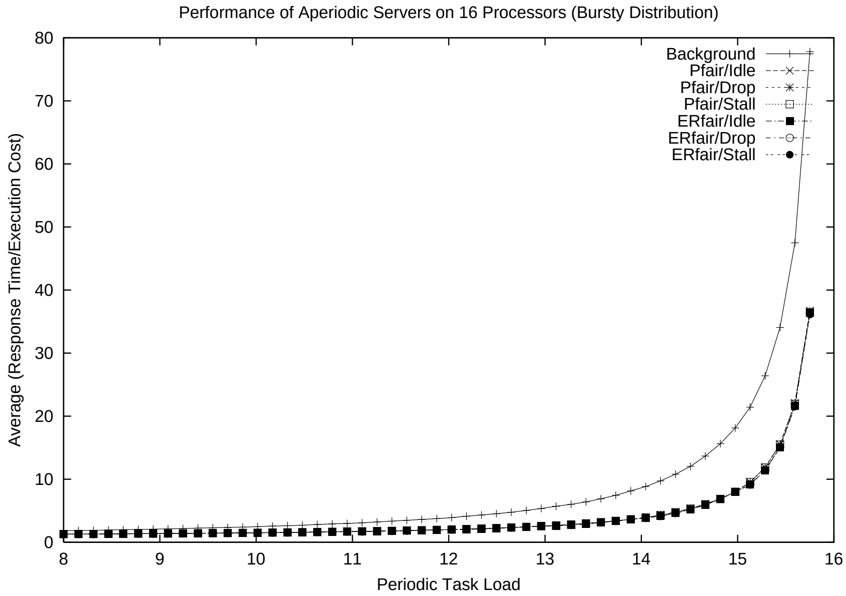



**(a)**

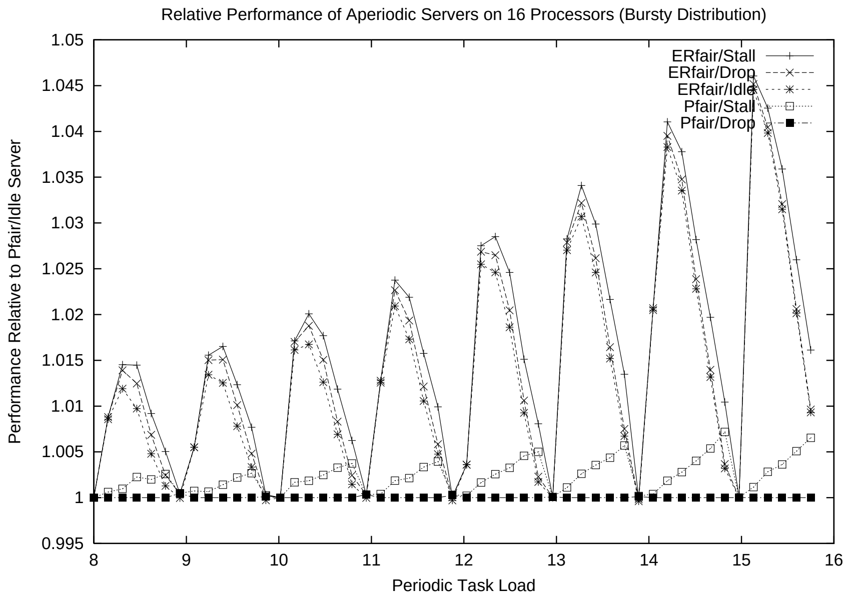



**(b)**

Figure 6.11: Simulation results on 16 processors with uniform distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

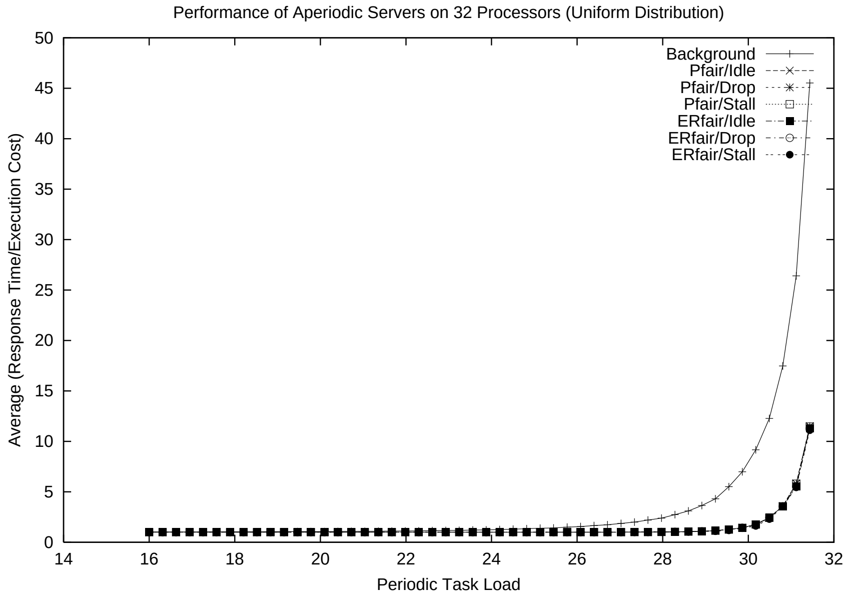



**(a)**

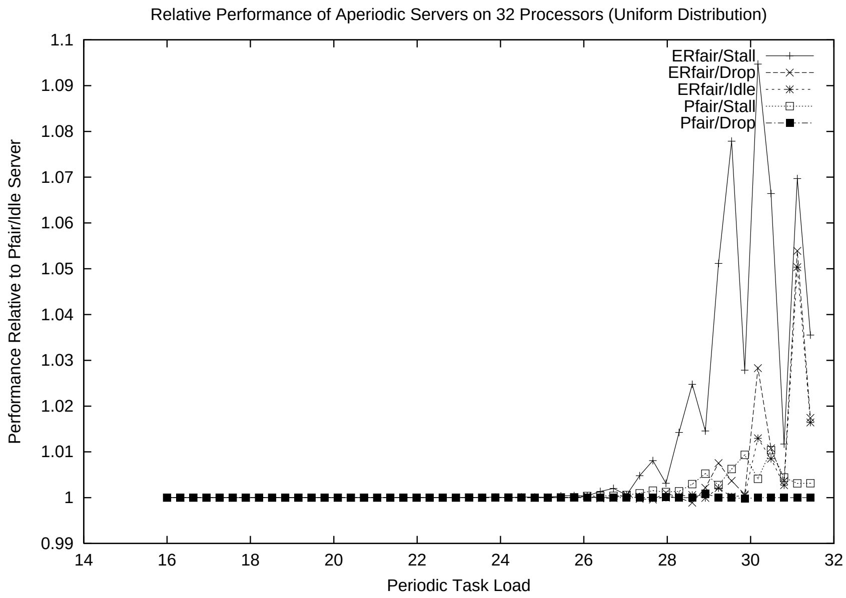



**(b)**

Figure 6.12: Simulation results on 16 processors with bursty distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

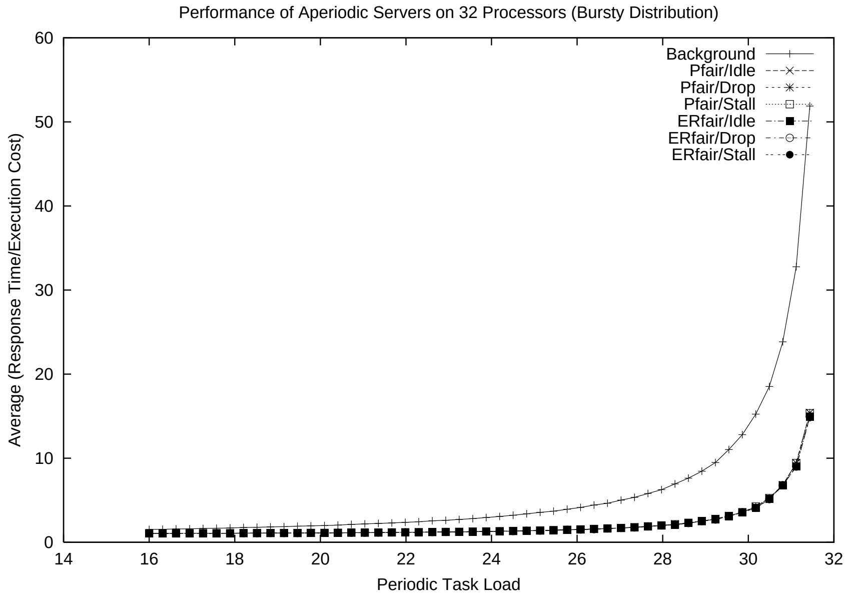



**(a)**

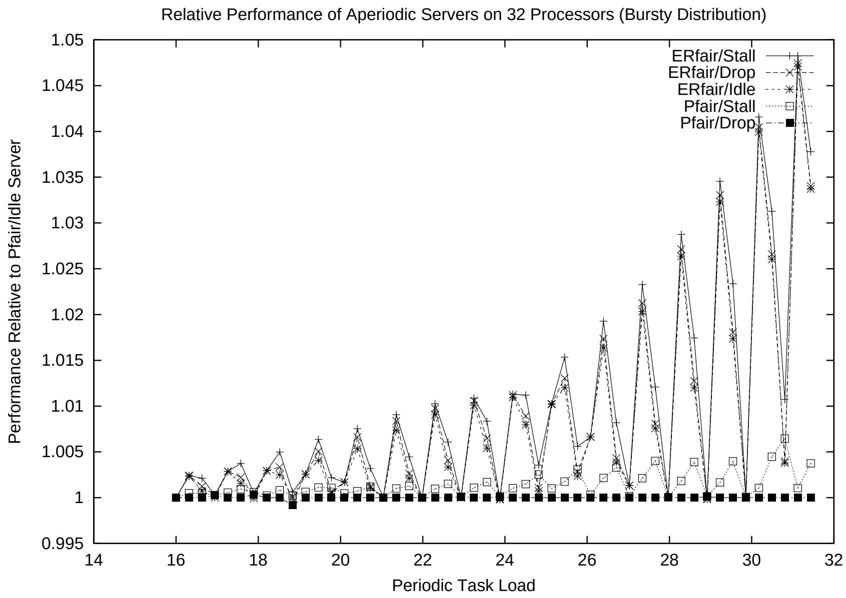



**(b)**

Figure 6.13: Simulation results on 32 processors with uniform distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.




**(a)**




**(b)**

Figure 6.14: Simulation results on 32 processors with bursty distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of each scheme. **(b)** Performance relative to an idling Pfair sever.

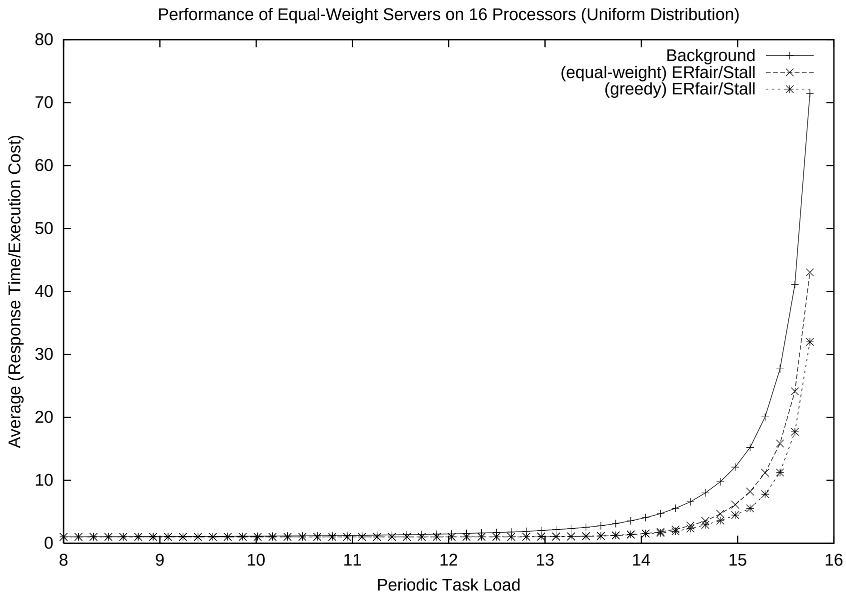
system (refer to inset (a) of Figures 6.13 and 6.14) the improvement over background scheduling approaches 250% in the bursty case, and 310% in the uniform case.

Note that the graphs in insets (b) of Figures 6.7–6.14 are quite different from their counterparts in Figure 6.5. This is because the number of servers here is sometimes more than one, and under the greedy policy, many of these servers will have a weight of one. Such a server is scheduled in every slot, and hence neither early-releasing nor stalling can improve its performance. When the periodic task load is integral, the greedy policy in fact results in *all* servers having a weight of one. This is why the various curves coincide at integral load levels. Intuitively, the difference between the implementations will be more pronounced when there are fewer unit-weight servers and when the non-unit-weight server's weight is around 0.5. When a server's weight is *too* small (much less than 0.5), its low utilization tends to keep the queue full, in which case the three variants perform similarly. Note that for 4 processors or more, ERfair/stall is up to 11% or 12% better than Pfair/drop, which is quite a bit more than for 2 processors. This suggests that the differences among the various schemes we have proposed are more pronounced on larger systems.

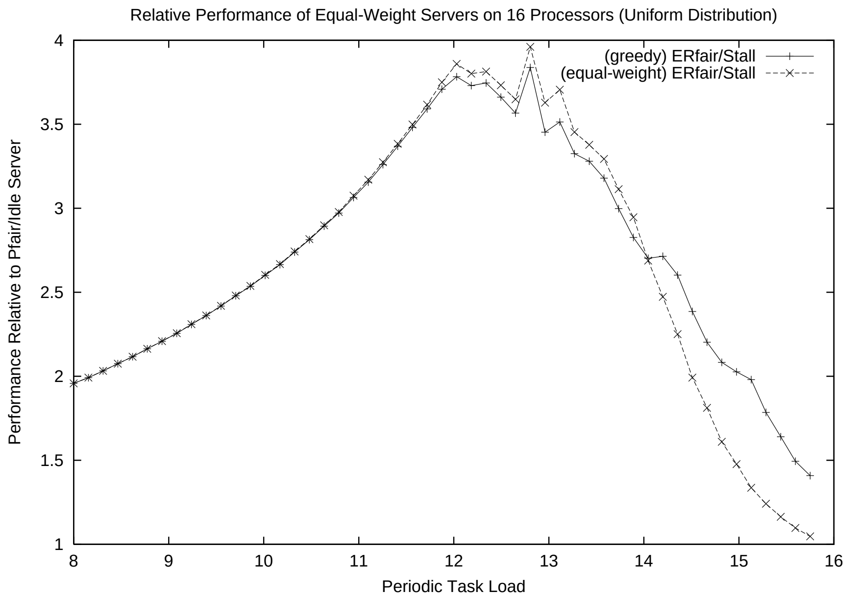
**Exception.** Note that, in inset (b) of Figures 6.9 and 6.11, ERfair/drop occasionally performs worse than Pfair/idle. This is because, when subtasks can become eligible earlier, there is a greater potential that a subtask is dropped (there are more slots where this could happen). Hence, response times suffer. Such a scenario is unlikely in the case of bursty releases since the aperiodic task queue is more likely to be backlogged.

**Comparison of greedy policy with equal-weight policy.** As mentioned in Section 6.2, one can devise several ways for choosing the number of servers and the weight distribution. We now describe results comparing the greedy policy with the following equal-weight policy: *any spare capacity is distributed uniformly among $M$ servers*, where $M$ is the number of processors. Figures 6.15 and 6.16 illustrate the difference in performance of the two approaches.

For clarity, only the results for ERfair/stall are shown. Figure 6.15 shows the results under uniform aperiodic task releases, while Figure 6.16 shows the results under bursty aperiodic task releases. As can be seen from both figures, the performance of the servers under the greedy policy is better than under the equal-weight policy. As seen in Figure 6.16(b), the difference is more pronounced if the aperiodic task releases are bursty. The main reason is that in the case of bursty releases, the servers are

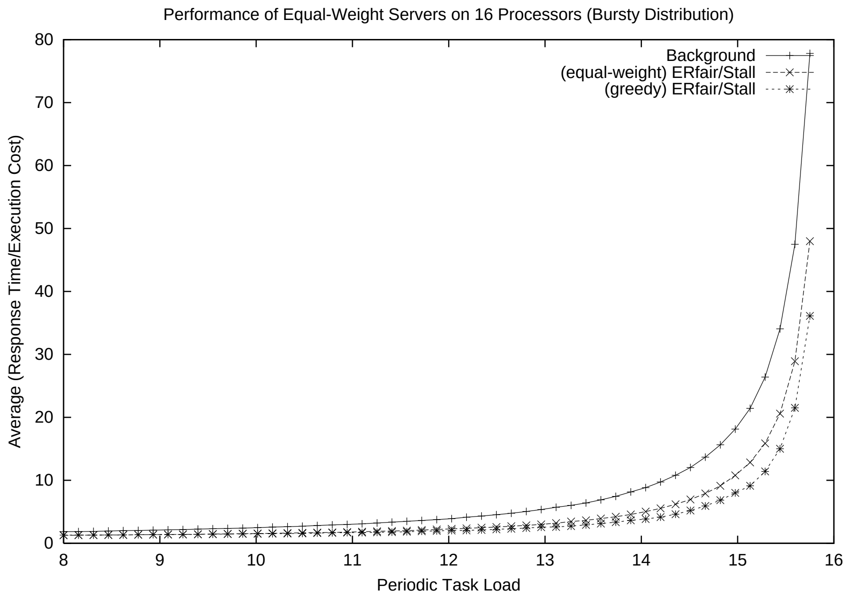



**(a)**

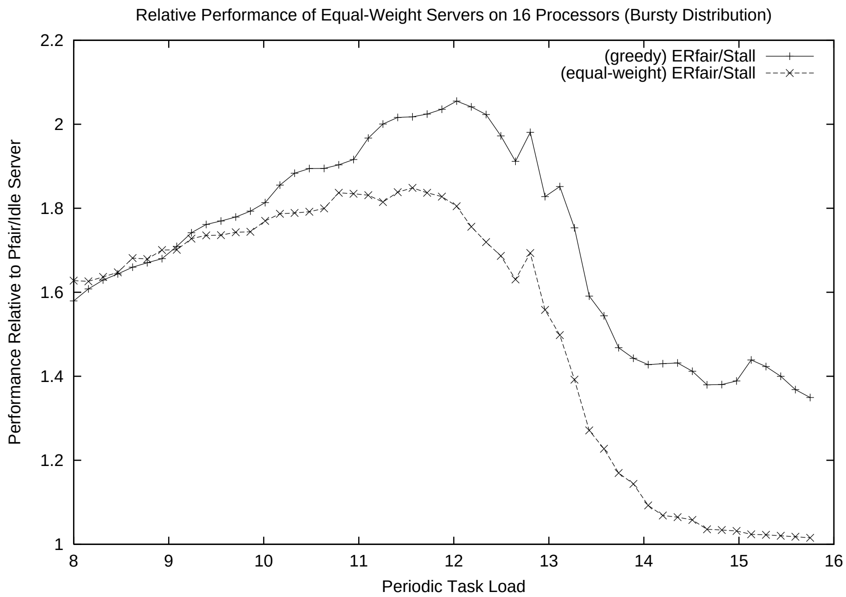



**(b)**

Figure 6.15: Simulation results on 16 processors with uniform distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of the stalling ERfair servers under both greedy and equal-weight policy. **(b)** Performance of the two stalling ERfair servers relative to an idling Pfair sever.




**(a)**




**(b)**

Figure 6.16: Simulation results on 16 processors with bursty distribution of aperiodic task releases. **(a)** Actual performance (*i.e.*, normalized response times) of the stalling ERfair servers under both greedy and equal-weight policy. **(b)** Performance of the two stalling ERfair servers relative to an idling Pfair sever.

backlogged most of the time and hence effectively act as a periodic task. As discussed in Section 6.2, the service received under the greedy policy is greater, which leads to better response times.

## 6.4 Summary

In this chapter, we have presented two server implementations for multiplexing aperiodic and recurrent real-time tasks in fair-scheduled multiprocessor systems. This is the first work to consider the problem of integrating support for aperiodic tasks within fair multiprocessor scheduling algorithms. We have also provided admission-control tests for the scheduling of hard aperiodic tasks, and have pointed out some additional complexities arising in server-based implementations on multiprocessors along with some ways to handle them. Most of these complexities arise because of the parallelism that exists in such systems. We have also provided experimental results that demonstrate the effectiveness of our implementations.

# Chapter 7

# Conclusions and Future Work

The Pfair scheduling approach introduced by Baruah *et al.* [BCPV96] is currently the only known way to optimally schedule recurrent real-time tasks on multiprocessors. Baruah *et al.* presented two Pfair algorithms PF [BCPV96] and PD [BGP95] that are optimal for scheduling periodic task systems on multiprocessors. Both algorithms prioritize subtasks by their deadlines; they differ in the way they break ties between equal subtask deadlines. Selecting appropriate tie-breaking rules is the most important concern in designing optimal Pfair algorithms.

In this dissertation, we have extended this initial work of Baruah *et al.* in several directions. In this chapter, we summarize our results and then briefly discuss directions for future research.

## 7.1 Summary

In Chapter 3, we presented the $PD^2$ algorithm, obtained by eliminating two of the tie-breaking rules of PD. (The $PD^2$ algorithm is the most efficient optimal Pfair algorithm known to date.) We also introduced the concept of ERfair scheduling, which is a work-conserving variant of Pfair scheduling. Using a swapping technique, we proved that the $PD^2$ algorithm is optimal for mixed Pfair/ERfair scheduling of asynchronous periodic task systems. This proof reveals several of the key insights involved in Pfair scheduling. Further, we showed that it is unlikely that more efficient optimal algorithms exist. In particular, we demonstrated through a collection of counterexamples that if any tie-breaking rule in $PD^2$ is eliminated or replaced by a more efficient rule, then periodic task systems exist that miss a deadline. Additionally, we presented results of experiments (involving randomly-generated task systems) that compare $PD^2$ and

EDF-FF, which is one of the commonly studied partitioning approaches. These results strongly indicate the viability of using $PD^2$ (and using Pfair scheduling algorithms, in general); $PD^2$ is more suitable for systems that support dynamic tasks or non-independent tasks, or that have heavy tasks.

In Chapter 4, we presented a new task model called the intra-sporadic (IS) model. The notion of an IS task generalizes the notion of a sporadic task by allowing subtasks to be released late. This generality is useful in several applications, especially, ones that process packets arriving over a network: even if packets are dispatched by the sender in a periodic manner, due to network congestion, packets can arrive late or in bursts. By allowing subtasks to be released late, and also allowing them to become eligible early, the IS model simplifies the handling of such occurrences. We presented a feasibility test for scheduling IS task systems on multiprocessors and also proved that the $PD^2$ algorithm is optimal for scheduling such tasks. The latter result was actually obtained as a corollary to a more general result involving dynamic task systems. In dynamic task systems, tasks are allowed to leave and join the system; such systems have become very common with the proliferation of gaming and virtual-reality applications. We derived sufficient join/leave conditions for scheduling dynamic task systems using $PD^2$, and also demonstrate the tightness of the conditions through counterexamples.

In Chapter 5, we presented results involving the earliest-pseudo-deadline-first (EPDF) algorithm, which is simpler and more efficient than $PD^2$. We obtained sufficient conditions under which EPDF is optimal and hence, preferable to $PD^2$. In particular, we proved that if the weight of each task is at most $\frac{1}{M-1}$, then EPDF can guarantee all deadlines. We also presented schedulability conditions for a variant of EPDF (with a different definition of deadline) that allow task weights to be larger than $\frac{1}{M-1}$. In addition, we considered scheduling of soft real-time systems using EPDF. Since occasional deadline misses are allowed in soft real-time systems, it is not necessary to use $PD^2$ in such systems. We proved that EPDF guarantees a tardiness of at most one quantum for task systems in which each task has a weight of at most $\frac{M+1}{2M-3}$. We also obtained similar restrictions on individual task utilizations for larger tardiness thresholds.

In Chapter 6, we presented several approaches for integrating the scheduling of aperiodic tasks with real-time recurrent tasks on fair-scheduled multiprocessors. In these approaches, the spare processor capacity (*i.e.*, the difference between number of processors and the total weight of the real-time IS tasks) is distributed among several aperiodic servers in a greedy manner. Each server is scheduled as a recurrent IS task and it, in turn, schedules the aperiodic tasks that are assigned to it. We derived bounds on the response times of the aperiodic tasks; these bounds can also be used to design an admission control test for aperiodic tasks with deadlines. We also presented results of simulations that compared our approaches with background scheduling (which is the only other approach known for scheduling aperiodic tasks on fair-scheduled multiprocessors.)

## 7.2 Future Work

In this section, we discuss some of the remaining challenges in fair scheduling on multiprocessors.

### 7.2.1 Quantum Size

One of the limitations of current work on Pfair scheduling (including the work in this dissertation) is that all tasks are scheduled using the same quantum size. Such a restriction may be undesirable, and in fact, impractical for certain systems. For example, if Pfair scheduling algorithms are used for scheduling packets on multi-link routers, then variable-sized quanta may be necessary. In general, allowing variable-sized quanta will cause Pfair scheduling algorithm to be non-optimal. Further research is needed to determine bounds on tardiness under such conditions (assuming a given maximum size for any quantum).

Even within fixed-length quantum scheduling, a problem of practical importance would be to determine an *optimal* quantum size. Note that with a smaller quantum, it is more likely that all execution requirements and periods will be a multiple of the quantum size. However, a smaller quantum also leads to larger scheduling and preemption overheads (refer to Section 3.5), which in turn causes a loss in schedulability. These trade-offs need to be analyzed to determine the quantum size that leads to maximum schedulability.

A related challenge is the design of good global scheduling algorithms that are *not* quantum-based. Recall from Chapter 2 that the worst-case achievable utilization using global EDF is just one for any number of processors [DL78]. In recent work [SB02], we developed a deadline-based algorithm called EDF-US ($\frac{m}{2m-1}$), which significantly improves upon EDF. EDF-US ($\frac{m}{2m-1}$) differs from EDF in a simple way: it statically assigns highest priority to tasks with utilization more than $\frac{m}{2m-1}$ and schedules the remaining tasks using EDF. We have shown that EDF-US ($\frac{m}{2m-1}$) correctly schedules on $m$ processors any task system that has a total utilization of at most $\frac{m^2}{2m-1}$. Further research is needed to determine whether a non-quantum-based algorithm with a higher worst-case achievable utilization exists.

### 7.2.2 Fair Distribution of Spare Capacity

In Section 2.2.3, we described work on uniprocessor GPS scheduling. Unlike the ideal scheduler underlying the concept of Pfairness, under GPS, any spare processor capacity is distributed fairly among all tasks. Chandra *et al.* [CAGS00] recently extended the concept of GPS to multiprocessors by introducing generalized multiprocessor scheduling (GMS). Under GMS, spare capacity is distributed according to the weights, but, with the additional constraint that the share of each task is at most one. Chandra *et al.* also presented an algorithm that approximates GPS and demonstrated its effectiveness through experiments. However, they did not formally analyze their approach.

Note that one way to utilize spare capacity is by task reweighting so that the total weight of the system always equals the number of processors. Since reweighting can be accomplished using leaves and joins, our results pertaining to dynamic task systems provide a first step towards obtaining formal proofs for GPS-based systems. However, there is clearly more scope for further research. In particular, it would be interesting to know what kind of bounds on tardiness can be derived when (J1) and (L1) (refer to Section 4.2) are used in conjunction with $\mathrm{PD}^2$. On the same note, further research is needed to determine tighter tardiness bounds for EPDF or to show that bounds given in Chapter 5 are tight.

In recent work [ABS03], we presented a new notion of multiprocessor fairness called *quick-release fair* (*QRfair*) scheduling, under which idle processor time is distributed fairly using heuristics. Under QRfair scheduling, each task is allowed to have both a minimum and a maximum weight. We presented a quick-release variant of $\mathrm{PD}^2$ called $\mathrm{PD}^{\mathrm{Q}}$, which schedules each task at a rate that is **(i)** at least that implied by its minimum weight and **(ii)** at most that implied by its maximum weight. We also presented results from extensive simulation experiments that show that $\mathrm{PD}^{\mathrm{Q}}$ allocates spare. Further research is needed to formally prove this property about $\mathrm{PD}^{\mathrm{Q}}$.

### 7.2.3 Tasks with Relative Deadlines less than Periods

Another issue that warrants further research is the consideration of periodic task systems in which relative deadlines are allowed to be different from periods. This generality allows us to model precedence constraints between tasks. Relative deadlines greater than periods can be handled by artificially reducing it to be equal to the period. We now briefly discuss feasibility issues regarding systems in which relative deadlines are allowed to be less than periods. (We restrict this discussion to synchronous periodic task systems; the feasibility problem for asynchronous periodic task systems with relative deadlines less than periods is known to be intractable even on a single processor [BHR93, LM80].)

**Exponential-time feasibility tests.** (We consider this issue in slightly more detail as some preliminary results have been obtained.) For uniprocessors, Baruah *et al.* [BHR93] presented the following exponential-time feasibility test: a task system $\tau$ is feasible on a uniprocessor if and only if

$$\text{for all } t \in [0, L), \sum_{T \in \tau} T.e \cdot \left( \left\lfloor \frac{t - T.d}{T.p} \right\rfloor + 1 \right) \leq t, \tag{7.1}$$

where $L$ is the LCM of the task periods. Such a test is called a demand-based test because $T.e \cdot \left( \left\lfloor \frac{t - T.d}{T.p} \right\rfloor + 1 \right)$ represents the processor time demanded by $T$ over the interval $[0, t)$.

The above test effectively simulates the behavior of the EDF algorithm and is based on the fact that EDF (a polynomial-time algorithm) is optimal for scheduling periodic tasks on multiprocessors, even if relative deadlines differ from periods. However, no such algorithm is known for multiprocessors. Still, an exponential-time feasibility test for multiprocessors can be obtained by a simple extension of the network flow technique presented in Chapter 4. To check the feasibility of a task system, we can build a corresponding network graph and verify whether a flow of the required size exists. However, this feasibility test has some limitations in the sense that it is not easy to obtain more efficient conditions for special cases. For instance, for uniprocessors, Baruah *et al.* [BHR93] presented a more efficient (pseudo-polynomial-time) test based on (7.1) that applies if the total utilization of the task system is strictly less than one.

Unfortunately, obtaining demand-based tests for multiprocessors is very difficult. As we now show, the following simple and intuitive extension of the uniprocessor demand-based feasibility test does not work on multiprocessors: a task system $\tau$ is feasible on

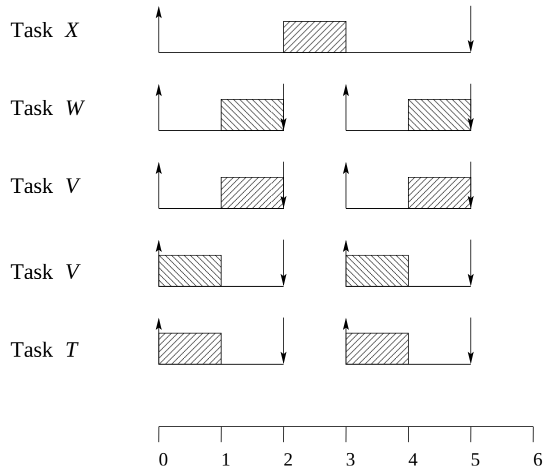



Figure 7.1: A two-processor schedule for a task system with five tasks $T$, $U$, $V$, $W$, and $X$ is shown. The up arrows corresponds to job releases and down arrows correspond to job deadlines. Each task is shown on a separate line.

$M$ processors if and only if for all $t \in [0, L)$, $\sum_{T \in \tau} T.e \cdot \left( \left\lfloor \frac{t - T.d}{T.p} \right\rfloor + 1 \right) \leq M \cdot t$.

Consider a two-processor system with five tasks: $T$, $U$, $V$, $W$, and $X$. Tasks $T$ through $W$ each have a period of 3, an execution requirement of 1, and a relative deadline of 2. Task $X$ has a period of 6, an execution requirement of 2, and a relative deadline of 5. Now, the demand over $[0, t)$ for any $t \in [0, 6]$ is as follows. (Refer to Figure 7.1.)

$t = 1$. Demand is 0, which is at most $1 \times 2$.

$t = 2$. Demand is 4, which is at most $2 \times 2$.

$t = 3$. Demand is 4, which is at most $3 \times 2$.

$t = 4$. Demand is 4, which is at most $4 \times 2$.

$t = 5$. Demand is 10, which is at most $5 \times 2$.

$t = 6$. Demand is 10, which is at most $6 \times 2$.

Thus, the test is satisfied, but the task system is not feasible (refer to Figure 7.1). To see why, note that over $[0, 2)$, the four jobs of $T$–$W$ must execute. This implies that over $[2, 3)$ only task $X$ can execute, and a processor is idle. Further, it can receive at most one unit of processor time. Thus, over the interval $[3, 5)$ each of the five tasks requires one unit and there are only four units available. Thus, one task will miss its deadline. The primary reason the demand-based test does not work on more than one processor is that it cannot identify slots in which a processor is idle.

**A simple polynomial-time schedulability test for PD$^2$.** An $O(N)$ (where $N$ is the total number of tasks) schedulability test can be easily obtained for PD$^2$ from the result in Corollary 4.5.2. Before presenting this test, we define a new term. The *density* of a task is the ratio of its execution requirement to its relative deadline; thus, the density of task $T$ is $\frac{T.e}{T.d}$. (Note that if $T.d = T.p$, then $T'$ density and weight are equal.) The *density of a task system* is the sum of the densities of all the tasks in that system.

Given any task system $\tau$ in which relative deadlines differ from periods, we can obtain an instance of sporadic task system $\tau'$ that has the same pattern of releases and deadlines. In particular, for each $T \in \tau$, there is a task $T' \in \tau'$ such that $T'.\phi = T.\phi$, $T'.e = T.e$, $T'.p = T.d$, $T'.d = T'.p$. If the density of $\tau$ is at most $M$, then the weight of $\tau'$ is at most $M$. By Corollary 4.5.2, $\tau'$ can be correctly scheduled using PD$^2$. Thus, $\tau$ can be correctly scheduled if its density is at most $M$.

However, this schedulability condition is very loose. In fact, there exist task systems with density greater than any given $n$ that are feasible of a single processor. For example, consider the following task system consisting of $k$ tasks numbered from 1 to $k$. Each task has a period of $k$, an execution requirement of 1 and a phase of 0; the $i^{th}$ task has a relative deadline of $i$. It is easy to see that in any period of $[(j-1)k, jk)$, a valid schedule can be obtained by executing tasks in the order 1 to $k$. Note that the density of this task system is $\sum_{i=1}^{k} \frac{1}{i}$. Since $\sum_{i=1}^{\infty} \frac{1}{i}$ is unbounded, there exists a task system that is feasible and has a density greater than $n$ for any given $n$.

The above discussion clearly indicates that more research is needed to obtain tighter efficient schedulability conditions.

# Appendix A

# Properties about Flows for IS Tasks

We now prove several properties that are used in the proofs in Section 4.3. With the exception of (B3) (see below), all of the properties below apply to GIS task systems.

**(L)** For a light task $T$, if $T_k$ is the successor of $T_i$, then $d(T_k) \geq d(T_i) + 2$.

*Proof.* Because $T_k$ is $T_i$'s successor, $k \geq i+1$. Hence, $d(T_k) \geq \theta(T_k) + \left\lceil \frac{i+1}{wt(T)} \right\rceil$. By (4.3), $d(T_k) \geq \theta(T_i) + \left\lceil \frac{i+1}{wt(T)} \right\rceil$. Therefore, by (4.2), $d(T_k) - d(T_i) \geq \left\lceil \frac{i+1}{wt(T)} \right\rceil - \left\lceil \frac{i}{wt(T)} \right\rceil$. Thus,

$$\begin{array}{rcll}
d(T_k) - d(T_i) & \geq & \frac{i+1}{wt(T)} - \left\lceil \frac{i}{wt(T)} \right\rceil & , \ \lceil x \rceil \geq x \\
& > & \frac{i+1}{wt(T)} - \frac{i}{wt(T)} - 1 & , \ \lceil x \rceil < x + 1 \\
& = & \frac{i}{wt(T)} - 1 & , \ \text{by simplification} \\
& = & \frac{1}{wt(T)} - 1 & , \ i \geq 1 \\
& > & 2 - 1 & , \ wt(T) < 1/2 \Rightarrow \frac{1}{wt(T)} > 2
\end{array}$$

Therefore, $d(T_k) > d(T_i) + 1$, *i.e.*, $d(T_k) \geq d(T_i) + 2$. ■

**(B1)** Let $T_i$ be a subtask with $b(T_i) = 1$. If $T_{i+1}$ exists, then $f(T_i, d(T_i) - 1) + f(T_{i+1}, r(T_{i+1})) = wt(T)$.

*Proof.* By (4.6), $f(T_i, d(T_i) - 1) = i - \left( \left\lceil \frac{i}{wt(T)} \right\rceil - 1 \right) \times wt(T)$, and $f(T_{i+1}, r(T_{i+1})) = \left( \left\lfloor \frac{i}{wt(T)} \right\rfloor + 1 \right) \times wt(T) - i$. Since $b(T_i) = 1$, by (2.11), $\left\lceil \frac{i}{wt(T)} \right\rceil = \left\lfloor \frac{i}{wt(T)} \right\rfloor + 1$. Hence, $f(T_{i+1}, r(T_{i+1})) = \left\lceil \frac{i}{wt(T)} \right\rceil \times wt(T) - i$. Therefore, $f(T_i, d(T_i)) + f(T_{i+1}, r(T_{i+1})) = wt(T)$. (See Figure 4.11.) ■

**(B2)** Let $T_i$ be a subtask such that $b(T_i) = 1$. If $T_{i+1}$ exists and is released late, *i.e.*, $r(T_{i+1}) \geq d(T_i)$, then $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, d(T_i)) \leq wt(T)$.

*Proof.* Because $T_{i+1}$ is released late, by (4.4), we have $r(T_{i+1}) \geq d(T_i)$. By (4.1) and (4.2), it follows that $r(T_k) > d(T_i)$ for all $k > i + 1$. Similarly, $d(T_j) < d(T_i)$ for all $j < i$. This implies that the slot $d(T_i) - 1$ lies within the PF-window of only one subtask, namely, $T_i$, and the slot $d(T_i)$ can lie within the PF-window of only one subtask, namely, $T_{i+1}$. Thus, the contribution to the flow in slot $d(T_i) - 1$ is $f(T_i, d(T_i) - 1)$ and the contribution to slot $d(T_i)$ is at most $f(T_{i+1}, r(T_{i+1}))$. Hence, by (B1), $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, d(T_i)) \leq wt(T)$. ■

**(B3)** If $T_i$ and $T_k$ are subtasks of an *IS* heavy task $T$ such that $k > i$ and $r(T_k) < D(T_i)$, then $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) \leq wt(T)$.

*Proof.* If $b(T_i) = 0$, then $D(T_i) = d(T_i)$. In this case, $r(T_k) \geq D(T_i)$ holds, since (4.1) implies $r(T_k) \geq d(T_i)$ (thus, no task $T_k$ exists such that $k > i$ and $r(T_k) < D(T_i)$). In the rest of the proof, we assume that $b(T_i) = 1$. Note that, by the definition of $D$ (the group deadline), because $r(T_k) < D(T_i)$, for all $j \in \{i+1, \ldots, k-1\}$, $|w(T_j)| = 2$ and $b(T_j) = 1$. Because $|w(T_j)| = 2$, we have $d(T_j) = r(T_j) + 2$. Because the total flow for a subtask is one, this implies that

$$\forall j \in \{i+1, \ldots, k-1\}, f(T_j, r(T_j)) + f(T_j, d(T_j) - 1) = 1. \tag{A.1}$$

For all $j \in \{i, \ldots, k-1\}$, $b(T_j) = 1$. Therefore, by (B1), $f(T_j, d(T_j)-1) + f(T_{j+1}, r(T_{j+1})) = wt(T)$. Therefore, $\sum_{j=i}^{k-1} f(T_j, d(T_j) - 1) + f(T_{j+1}, r(T_{j+1})) = (k - i) \times wt(T)$. Rewriting, we get $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) + \sum_{j=i+1}^{k-1} (f(T_j, r(T_j)) + f(T_j, d(T_j) - 1)) = (k - i) \times wt(T)$. By (A.1), this implies that

$$f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) + k - i - 1 = (k - i) \times wt(T).$$

Therefore, $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) = wt(T) + (k - i - 1)(wt(T) - 1)$. Because $k \geq i + 1$ and $wt(T) \leq 1$ for all $T$, we have $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) \leq wt(T)$. ■

**(B4)** Let $T_i$ be a subtask of a heavy GIS task $T$ and let $T_k$ $(k > i)$ be a subtask such that $r(T_k) < D(T_i)$. Then, $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) \leq wt(T)$.

*Proof.* Because $T$ is a GIS task, there is an IS task $U$ such that $wt(U) = wt(T)$, all subtasks between $U_i$ and $U_k$ are present, and $r(U_k) = r(T_k)$. Hence, $r(U_k) < D(U_i)$.

By (B3), $f(U_i, d(U_i) - 1) + f(U_k, r(U_k)) \leq wt(U)$. Corresponding subtasks in $T$ and $U$ have identical flows. Thus, $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) \leq wt(T)$. ■

**(F2)** Let $T_i$ be a subtask of a GIS task and let $T_k$ be its successor. If $b(T_i) = 1$ and $r(T_k) \geq d(T_i)$, then $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, d(T_i)) \leq wt(T)$.

*Proof.* If $k = i + 1$, then by (B2), $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, d(T_i)) \leq wt(T)$. Also, if $r(T_k) > d(T_i)$, then $\mathit{flow}(T, d(T_i)) = 0$. Hence, by (F1), $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, d(T_i)) \leq wt(T)$.

In the rest of the proof, we assume that $k > i + 1$ and $r(T_k) = d(T_i)$. We first show that $T$ must be heavy. If $T$ is light, then by (L), we have $d(T_{i+1}) > d(T_i) + 1$. By (4.4), we also have $r(T_k) \geq d(T_{i+1}) - 1$ and therefore, $r(T_k) > d(T_i)$, which contradicts $r(T_k) = d(T_i)$.

Thus, $T$ is heavy. Because $b(T_i) = 1$, by the definition of $D$, $D(T_i) > d(T_i)$. Hence, because $r(T_k) = d(T_i)$, we have $r(T_k) < D(T_i)$. Thus, by (B4), $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, r(T_k)) \leq wt(T)$. Because $r(T_k) = d(T_i)$, we have $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, d(T_i)) \leq wt(T)$. ■

**(F3)** Let $T_i$ be a subtask of a heavy GIS task $T$ such that $b(T_i) = 1$ and let $T_k$ be the successor of $T_i$. If $u \in \{d(T_i), \ldots, D(T_i) - 1\}$ and $u \leq r(T_k)$, then $\mathit{flow}(T, d(T_i)) + \mathit{flow}(T, u) \leq wt(T)$.

*Proof.* Since $b(T_i) = 1$, by the definition of $D$, $D(T_i) > d(T_i)$. Since $u \geq d(T_i)$ and $T_k$ is $T_i$'s successor, if $r(T_k) > u$, then $\mathit{flow}(T, u) = 0$. Thus, by (F1), $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, u) \leq wt(T)$. The other possibility is $r(T_k) = u$, which implies $r(T_k) < D(T_i)$. In this case, by (B4), $f(T_i, d(T_i) - 1) + f(T_k, r(T_k)) \leq wt(T)$. Thus, $\mathit{flow}(T, d(T_i) - 1) + \mathit{flow}(T, u) \leq wt(T)$. ■

# Bibliography


[ABJ99] J. Anderson, S. Baruah, and K. Jeffay. Parallel switching in connection-oriented networks. In *Proceedings of the 20th IEEE Real-time Systems Symposium*, pages 200–209, December 1999.

[ABS03] J. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In *Proceedings of the 24th IEEE Real-time Systems Symposium*, December 2003. To appear.

[AHKB00] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[AS00a] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 35–43, June 2000.

[AS00b] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the 7th International Conference on Real-time Computing Systems and Applications*, pages 297–306, December 2000.

[AS01] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-time Systems*, pages 76–85, June 2001.

[AS04] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer Systems and Sciences*, 2004. To appear. (Most of the results in this paper were presented in preliminary form at the 12th and 13th Euromicro Conferences on Real-time Systems.).

[Bar95] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the 16th IEEE Real-time Systems Symposium*, pages 200–209, December 1995.

[Bar00] S. Baruah. Scheduling periodic tasks on uniform multiprocessors. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 7–14, June 2000.

[Bar01] S. Baruah. Scheduling periodic tasks on uniform multiprocessors. *Information Processing Letters*, 80(2):97–104, 2001.

[BCPV96] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[BFP+73] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer System and Sciences*, 7:448–461, 1973.

[BGP95] S. Baruah, J. Gehrke, and C. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[BGP+97] S. Baruah, J. Gehrke, C. Plaxton, I. Stoica, H. Abdel-Wahab, and K. Jeffay. Fair on-line scheduling of a dynamic set of tasks on a single resource. *Information processing Letters*, 64(1):43–51, October 1997.

[BHR93] S. Baruah, R. Howell, and L. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[BLOS95] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. Assigning real-time tasks to homogeneous multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.

[BR02] E. Bampis and G. N. Rouskas. The scheduling and wavelength assignment problem in optical wdm networks. *IEEE/OSA Journal of Lightwave Technology*, 20(5):782–789, 2002.

[BZ96] J. Bennett and H. Zhang. WF$^2$Q: Worst-case fair queueing. In *Proceedings of IEEE INFOCOM*, pages 120–128, March 1996.

[BZ97] J. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[CAGS00] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of the 4th ACM Symposium on Operating System Design and Implementation*, pages 45–58, October 2000.

[CAS01] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proceedings of the 7th IEEE Real-time Technology and Applications Symposium*, pages 3–14, May 2001.

[CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[CL90] M. Chen and K. Lin. Dynamic priority ceiling: A concurrency control protocol for real time systems. *Real-time Systems*, 2(1):325–346, 1990.

[CO01] K. Coffman and A. Odlyzko. The size and growth rate of the internet. March 2001. Available at `http://www.firstmonday.dk/issues/issue3_10/coffman/`.

[DD86] S. Davari and S. Dhall. An on-line algorithm for real-time tasks allocation. In *Proceedings of the 7th Real-time Systems Symposium*, pages 194–200, 1986.

[Der74] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 1–12, 1989.

[DL78] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[DLS97] Z. Deng, J. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of 9th Euromicro Workshop on Real-time Systems*, pages 191–199, June 1997.

[DM89] M. Dertouzos and A. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.

[FF62] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FGP+00] A. Ferrari, S. Garue, M Peri, S. Pezzini, L.Valsecchi, F. Andretta, and W. Nesci. The design and implementation of a dual-core platform for power-train systems. In *Convergence 2000*, October 2000.

[GB95] T. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Real-time Systems*, 9(1):31–67, 1995.

[GFB03] 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.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Gol94] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of IEEE INFOCOM*, pages 636–646, April 1994.

[GVC96] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM Sigcomm*, August 1996.

[HA01] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, December 2001.

[HA02a] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 149–158, December 2002.

[HA02b] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pages 111–120, June 2002.

[HW79] G. Hardy and E. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 1979.

[JB95] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 64–75, April 1995.

[Jef92] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. In *Proceedings of the 13th IEEE Symposium on Real-time Systems*, pages 89–98, December 1992.

[JG99] K. Jeffay and S. Goddard. The rate-based execution model. In *Proceedings of the 20th IEEE Real-time Systems Symposium*, pages 304–314, December 1999.

[JSMA98] K. Jeffay, F. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 480–491, December 1998.

[LGDG00] J. Lopez, M. Garcia, J. Diaz, and D. 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 25–33, June 2000.

[LHS+98] C.-G. Lee, J. Hahn, Y.-M. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLH+01] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, 2001.

[LM80] J. Leung and M. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[LMM98] S. Lauzac, R. Melhem, and D. Mosse. An efficient RMS admission control and its application to multiprocessor scheduling. In *Proceedings of the International Symposium on Parallel Processing*, pages 511–518, March 1998.

[LRT92] J. Lehoczky and S. Ramos-Thuel. An optimal algoririthm for scheduling soft-aperiodic tasks in fixed priority preemptive systems. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 110–123, December 1992.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th IEEE Real-time Systems Symposium*, pages 166–171, December 1989.

[LSS87] J. Lehoczky, L. Sha, and J. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of 8th IEEE Real-time Systems Symposium*, pages 261–270, 1987.

[Mok83] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-real-time Environment*, 1983. Ph.D. thesis, Massachussets Institute of Technology.

[MR99] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-time Systems Symposium*, pages 294–303, December 1999.

[OB98] D. Oh and T. Baker. Utilization bounds for $N$-processor rate monotone scheduling with static processor asssignment. *Real-time Systems*, 15(2):183–192, 1998.

[OS95a] Y. Oh and S. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-time Systems*, 9(3):207–239, 1995.

[OS95b] Y. Oh and S. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical Report 95-16, Department of Computer Science, University of Virginia, March 1995.

[PG93] A. Parekh and R. Gallagher. A generalized processor sharing approach to flow-control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[PG94] A. Parekh and R. Gallagher. A generalized processor sharing approach to flow-control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[Raj02] R. Rajkumar. Personal communication. 2002.

[Ram97] S. Ramamurthy. *A Lock-Free Approach to Object Sharing in Real-time Systems*, 1997. Ph.D. thesis, University of North Carolina at Chapel Hill.

[RSL88] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-time Systems Symposium*, pages 259–269. IEEE, 1988.

[SA02] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[SA03] 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.

[SA04a] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. *Journal of Embedded Computing*, June 2004. Under submission. (A preliminary version of this paper was presented at the 15th Euromicro Conference on Real-time Systems.).

[SA04b] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 2004. Under submission. (A preliminary version of this paper was presented at the 11th International Workshop on Parallel and Distributed Real-time Systems.).

[SAWJ+96] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-time Systems Symposium*, pages 288–299, December 1996.

[SB96] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-time Systems*, 10(2):179–210, 1996.

[SB02] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, November 2002.

[SHA02] A. Srinivasan, P. Holman, and J. Anderson. Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pages 19–28, June 2002.

[SHA+03] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *Proceedings of the 2nd Workshop on Network Processors*, pages 48–62, February 2003.

[SHAB03] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of the 11th International Workshop on Parallel and Distributed Real-time Systems*, April 2003.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SRLR89] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven pre-emptive scheduling. *Real-time Systems*, 1(3):244–264, 1989.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-time Systems*, 1(1):27–60, 1989.

[SVC98] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *Proceedings of the 10th Euromicro Workshop on Real-time Systems*, pages 53–60, June 1998.

[SZN97] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *Proceedings of ACM Sigcomm*, August 1997.

[TBW92] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority pre-emptive scheduled systems. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 100–109, December 1992.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–315, 1978.

[WH95] D. Wood and M. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, 1995.

[Zha91] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.