Soft Real-Time Scheduling on Multiprocessors

by
UmaMaheswari C. Devi

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
2006

Approved by:

Prof. James H. Anderson

Prof. Sanjoy K. Baruah

Prof. Kevin Jeffay

Prof. Daniel Mossé

Prof. Ketan Mayer-Patel

Prof. Jasleen Kaur

Abstract

UMAMAHESWARI C. DEVI: Soft Real-Time Scheduling on Multiprocessors.
(Under the direction of Prof. James H. Anderson.)

The design of real-time systems is being impacted by two trends. First, tightly-coupled multiprocessor platforms are becoming quite common. This is evidenced by the availability of affordable symmetric shared-memory multiprocessors and the emergence of multicore architectures. Second, there is an increase in the number of real-time systems that require only soft real-time guarantees and have workloads that necessitate a multiprocessor. Examples of such systems include some tracking, signal-processing, and multimedia systems. Due to the above trends, cost-effective multiprocessor-based soft real-time system designs are of growing importance.

Most prior research on real-time scheduling on multiprocessors has focused only on hard real-time systems. In a hard real-time system, no deadline may ever be missed. To meet such stringent timing requirements, all known theoretically optimal scheduling algorithms tend to preempt process threads and migrate them across processors frequently, and also impose certain other restrictions. Hence, the overheads of such algorithms can significantly reduce the amount of useful work that is accomplished and limit their practical implementation. On the other hand, non-optimal algorithms that are more practical suffer from the drawback that their validation tests require workload restrictions that can approach roughly 50% of the available processing capacity. Thus, for soft real-time systems, which can tolerate occasional or bounded deadline misses, and hence, allow for a trade-off between timeliness and improved processor utilization, the existing scheduling algorithms or their validation tests can be overkill. The thesis of this dissertation is:

Processor utilization can be improved on multiprocessors while providing non-trivial soft real-time guarantees for different soft real-time applications, whose preemption and migration overheads can span different ranges and whose tolerances to tardiness are different, by designing new algorithms, simplifying optimal algorithms, and

developing new validation tests.

The above thesis is established by developing validation tests that are sufficient to provide soft real-time guarantees under non-optimal (but more practical) algorithms, designing and analyzing a new restricted-migration scheduling algorithm, determining the guarantees on timeliness that can be provided when some limiting restrictions of known optimal algorithms are relaxed, and quantifying the benefits of the proposed mechanisms through simulations.

First, we show that both preemptive and non-preemptive global *earliest-deadline-first*(EDF) scheduling can guarantee bounded tardiness (that is, lateness) to every recurrent real-time task system while requiring no restriction on the workload (except that it not exceed the available processing capacity). The tardiness bounds that we derive can be used to devise validation tests for soft real-time systems that are EDF-scheduled.

Though overheads due to migrations and other factors are lower under EDF (than under known optimal algorithms), task migrations are still unrestricted. This may be unappealing for some applications, but if migrations are forbidden entirely, then bounded tardiness cannot always be guaranteed. Hence, we consider providing an acceptable middle path between unrestricted-migration and no-migration algorithms, and as a second result, present a new algorithm that restricts, but does not eliminate, migrations. We also determine bounds on tardiness that can be guaranteed under this algorithm.

Finally, we consider a more efficient but non-optimal variant of an optimal class of algorithms called *Pfair* scheduling algorithms. We show that under this variant, called *earliest-pseudo-deadline-first* (EPDF) scheduling, significantly more liberal restrictions on workloads than previously known are sufficient for ensuring a specified tardiness bound. We also show that bounded tardiness can be guaranteed if some limiting restrictions of optimal Pfair algorithms are relaxed.

The algorithms considered in this dissertation differ in the tardiness bounds guaranteed and overheads imposed. Simulation studies show that these algorithms can guarantee bounded tardiness for a significant percentage of task sets that are not schedulable in a hard real-time sense. Furthermore, for each algorithm, conditions exist in which it may be the preferred choice.

Acknowledgments

My entry to graduate school and successful completion of this dissertation and the Ph.D. program are due to the confluence of some fortuitous happenings and the support and goodwill of several people. The following is my attempt at acknowledging everyone I am indebted to.

I am profoundly grateful to my advisor, Jim Anderson, for educating and guiding me over the past few years with great care, enthusiasm, and patience. Though I can fill pages thanking Jim, I will limit to only a couple of paragraphs. Foremost, I am thankful to Jim for making me consider doing a Ph.D. and taking me under his fold when I decided to go for it. Ever since, it has been an extreme pleasure and a privilege working for Jim and learning from him. Jim reposed a lot of confidence in me, which, I should confess, was at times overwhelming, and gave me enormous freedom in my work, all the while ensuring that I was making adequate progress. He helped relieve much of the tedium, assuage my apprehensions, boost my self-esteem, and make the whole endeavor a joy by being readily accessible, letting me have his undivided attention most of the time I walked in to his office, offering sound and timely advice, and when needed, suggesting corrective measures. His willingness for short, impromptu discussions—over a half-baked idea, or a new result, a fresh insight, or a concern, or just a recently-read paper—and provide his perspective, was much appreciated. I cannot help remarking that I have been amazed many a time at Jim's sharpness of mind and intellect, ability to effectively balance conflicting demands under various circumstances, thoroughness, sense of humor, and above all, genuine care and concern for his students.

I would like to thank Jim in particular for being patient with some of my sloppy writing, getting those fixed, and in the process, teaching me to write. His prompt and careful feedback on drafts served as a catalyst that accelerated writing and is perhaps a reason why his students tend to write the long dissertations that they are known for! Thanks are also due to Jim for his phenomenal support, which far exceeded what anyone can ever ask for, when I was in the academic job market. Finally, I cannot omit mentioning the numerous conference trips, five of which were to Europe, which Jim sponsored, and which have helped in widening my perspective on several aspects.

I feel honored to have had some other respected researchers also take the time to serve on my committee. In this regard, thanks are due to Sanjoy Baruah, Kevin Jeffay, Daniel Mossé, Ketan-Mayer Patel, and Jasleen Kaur. I am thankful to my entire committee for their feedback on my work and their flexibility in accommodating my requests while scheduling proposals and exams. Profound thanks are due to Sanjoy for his support and encouragement during my stay here. Sanjoy's work has inspired me a lot and he has influenced me to a good extent. Coincidentally, it turns out that but for Sanjoy, I would not have received admission to UNC! Special thanks are also due to Kevin for his encouragement and his concern and efforts that we receive a well-rounded education, and to Daniel for his detailed comments on my dissertation and taking the time to fly in and attend my defense in person. I am additionally indebted to Sanjoy, Kevin, and Daniel for writing me reference letters. Ketan and Jasleen have also been very supportive overall, and special thanks to Jasleen for her friendship and for sharing some of her interviewing experiences.

Thanks also go to IBM, and, in particular, to Andy Rindos, for their Ph.D. fellowship, which funded my final two years of study. Profs. Giuseppe Lipari and Al Mok wrote me reference letters, which is gratefully acknowledged. I am thankful to the entire faculty of UNC's Computer Science Department for the congenial and stimulating atmosphere that they help create. Special thanks to everyone from whom I have taken some excellent courses, and to Profs. Gary Bishop, Dinesh Manocha, Russ Taylor, and Henry Fuchs for willingly taking the time to help me acquire some academic-job interviewing skills. I owe it to Prof. David Stotts for funding my first year of study.

My work has benefitted to a good extent from the weekly real-time lunch meetings and the interactions I have had with past and present real-time systems students. I am grateful to Anand Srinivasan and Phil Holman for patiently clarifying some of my misconceptions during my formative days and helping me with my ramp up. The foundation for much of my work was laid by Anand in his dissertation (as will be evidenced by the numerous references), and I am thankful to both Anand and Phil for setting high standards in research and writing. Special thanks are due to Shelby Funk for her friendship and moral support. I am also very thankful for the support, friendship, and constructive criticism that I have received from Aaron Block, Nathan Fisher, John Calandrino, Hennadiy Leontyev, Abhishek Singh, Vasile Bud, Sankar Vijayaraghavan, Mithun Arora, and Billy Saelim. Special thanks go to Aaron, John, Hennadiy, and Vasile for their cooperation when we co-authored papers. Thanks are also due to the following DiRT friends: Jay Aikat, Sushanth Rewaskar, and Alok Shriram. I am especially thankful to Jay for her overall support and for taking the trouble to attend several of my practice talks and offer constructive feedback.

I would like to take this opportunity to extend my thanks to the administrative and technical staff of the Computer Science Department, as well, for providing us with an effective work environment, and for their readiness and cheer in attending to our needs. Special thanks in this regard go to Janet Jones, Karen Thigpen, Tammy Pike, Sandra Neely, Murray Anderegg, Charlie Bauserman, Linda Houseman, and Mike Stone.

I am fortunate to have been blessed with a loving and supportive family, who repose great trust in me despite not entirely approving my ways. I owe it to my mother and late grandfathers for instilling in me a passion for learning, and to my father for his pragmatism and for enlivening even mundane things through his wit and sense of humor. I am thankful to my sister and brother-in-law for their affection, and to my brother for his friendship and being someone I can turn to for almost anything. I am also thankful to my mother-in-law for her concern for me and her complete faith in me despite not knowing what I really do.

Above all, I am indebted in no small measure to my husband for having endured a lot during the past five years with only a few complaints. He put up with separation for several months, leftover food, and at times, an unkept home. But for his cooperation, patience, love, and faith, I would not have been able to continue with the Ph.D. program, let alone complete it successfully. I owe almost everything to him and hope to be able to repay him in full in the coming years.

Finally, I am thankful to God Almighty for the turn of events that led to this least-expected but valuable and rewarding phase of my life: most of what happened, starting with how I applied to grad school, was by chance and not due to any careful planning on my part.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BF	Boundary Fair
BWP	Blue When Possible
CBS	Constant Bandwidth Server
DBP	Distance Based Priority
DSP	Digital Signal Processing
DTMC	Discrete Time Markov Chain
EDF	Earliest Deadline First
EDF-fm	EDF with Fixed and Migrating tasks
EDF-P-NP	EDF with Preemptive and Non-Preemptive segments
EPDF	Earliest Pseudo-Deadline First
ER	Early Release
FIFO	First In First Out
g-EDF	global, preemptive EDF
GIS	Generalized Intra-Sporadic
HUF	Highest Utilization First
IS	Intra-Sporadic
ISR	Interrupt Service Routine
LEF	Lowest Execution (cost) First
LLF	Least Laxity First
LUF	Lowest Utilization First
MI	Miss Initiator
g-EDF	global, preemptive EDF
LITMUS$^{\mathrm{RT}}$	LInux Testbed for MUltiprocessor Scheduling in Real-Time Systems
PD	Pseudo-Deadline
pdf	probability density function
p-EDF	partitioned EDF
PF	PFair
PS	Processor Sharing (schedule)
QoS	Quality-of-Service
r-EDF	restricted-migration EDF

RM	Rate Monotonic
RM	Rational Rate Monotonic
RTO	Red Tasks Only
SMI	Successor of Miss Initiator
SMP	Symmetric (Shared-Memory) Multiprocessor
SRMS	Statistical Rate-Monotonic Scheduling
VR	Virtual Reality
WCET	Worst-Case Execution Time
WM	Weight Monotonic

Chapter 1

Introduction

The goal of this dissertation is to extend the theory of real-time scheduling to facilitate resource-efficient implementations of soft real-time systems on multiprocessors. This work is necessitated by the proliferation of both multiprocessor platforms and applications with workloads that warrant more than one processor and for which soft real-time guarantees are sufficient. This chapter begins with an introduction to real-time systems followed by a discussion of the subject matter of this dissertation. Needed background on real-time concepts, including real-time scheduling on multiprocessors, is then provided. Soft real-time systems are described next, following which motivation is provided for further research to support such systems on multiprocessors, and the thesis of this dissertation is stated. The chapter concludes by summarizing the contributions that this dissertation makes in support of the thesis and providing a roadmap for the rest of the dissertation.

1.1 What is a Real-Time System?

The distinguishing characteristic of a *real-time system* in comparison to a non-real-time system is the inclusion of *timing requirements* in its specification. That is, the correctness of a real-time system depends not only on logically correct segments of code that produce logically correct results, but also on executing the code segments and producing correct results within specific time frames. Thus, a real-time system is often said to possess dual notions of correctness, *logical* and *temporal*. Process control systems, which multiplex several control-law computations, radar signal-processing and tracking systems, and air traffic control systems are some examples of real-time systems.

Timing requirements and constraints in real-time systems are commonly specified as deadlines within which activities should complete execution. Consider a radar tracking system as an example. To track targets of interest, the radar system performs the following high-level activities or tasks: sends radio pulses towards the targets, receives and processes the echo signals returned to determine the position and velocity of the objects or sources that reflected the pulses, and finally, associates the sources with targets and updates their trajectories. For effective tracking, each of the above tasks should be invoked repeatedly at a frequency that depends on the distance, velocity, and the importance of the targets, and each invocation should complete execution within a specified time or *deadline*.

Another characteristic of a real-time system is that it should be *predictable*. Predictability means that it should be possible to show, demonstrate, or prove that requirements are *always* met subject to any assumptions made, such as on workloads [112]. In this dissertation, we focus on *a priori* ensuring that timing requirements are met; ensuring that non-timing requirements are met is out of the purview of this research.

Hard and soft real-time. Based on the cost of failure associated with not meeting them, timing constraints in real-time systems can be classified broadly as either *hard* or *soft*. A hard real-time constraint is one whose violation can lead to disastrous consequences such as loss of life or a significant loss to property. Industrial process-control systems and robots, controllers for automotive systems, and air-traffic controllers are some examples of systems with hard real-time constraints. In contrast, a soft real-time constraint is less critical; hence, soft real-time constraints can be violated. However, such violations are not desirable, either, as they may lead to degraded quality of service, and it is often the case that the extent of violation be bounded. Multimedia systems and virtual-reality systems are some examples of systems with soft real-time constraints.

Three important aspects or components in real-time system design are: *real-time system models* including task models and resource models; *scheduling algorithms*, which determine how the hardware resources are shared among the system's threads and/or processes; and *validation tests* that determine whether a real-time system's timing constraints will be met by a specified scheduling algorithm. Before considering these aspects in detail, we provide a high-level overview of the research addressed in this dissertation. The ensuing overview is intended to help place the background material covered later in proper perspective.

1.2 Dissertation Focus

As mentioned in the beginning of this chapter, the focus of this dissertation is to improve cost effectiveness while instantiating soft real-time systems on multiprocessors. The need for work in this direction is described below.

1.2.1 Motivation

One evident trend in the design of both general-purpose and embedded computing systems is the increase in the use of multiple processing elements that are tightly coupled. In the general-purpose arena, this is evidenced by the availability of affordable symmetric multiprocessor platforms (SMPs), and the emergence of multicore architectures. In the special-purpose and embedded arena, examples of multiprocessor designs include network processors, which are used for packet-processing tasks in programmable routers, system-on-chip platforms for multimedia processing in set-top boxes and digital TVs, and automotive power-train systems. If the current shift towards multicore architectures by prominent chip manufacturers such as Intel [2] and AMD [1] is any indication, then in the future, the standard computing platform in many settings can be expected to be a multiprocessor, and multiprocessor-based software designs will be inevitable. The need for multiprocessors is due to both architectural issues that impose limits on the performance that a single processing unit can deliver, and the prevalence of, and ever-increasing need for, higher computing demand from applications. Further, ideally, the energy consumed by an M-processor system is lower than that consumed by a single-processor system of equivalent capacity by a factor of approximately M^2 [8].

Some embedded systems, such as set-top boxes and automotive systems, are inherently real-time. Also, a number of emerging real-time applications exist that are instantiated on general-purpose systems and have high workloads. Systems that track people and machines, virtual-reality and computer-vision systems, systems that host web-sites, and some signal-processing systems are a few examples. Timing constraints in several of these embedded- and general-purpose-system applications are predominantly soft. Hence, with the shift towards multiprocessors, the need arises to instantiate soft real-time applications on multiprocessors.

1.2.2 Research Need and Overview

Minimizing hardware resource requirements is essential for realizing cost-effective system implementations. One way of minimizing resource requirements is through the careful management and allocation, *i.e.*, *scheduling*, of resources to competing requests. Relevant prior work on uniprocessor and multiprocessor-based soft real-time systems is discussed in Chapter 2. As can be seen from the discussion there, research on soft real-time scheduling on multiprocessors has been extremely limited: most prior research on real-time scheduling on multiprocessors has been confined to hard real-time systems, while that on soft real-time scheduling has been confined to uniprocessors. Uniprocessor scheduling theory does not readily generalize to multiprocessors, and basing soft real-time multiprocessor system designs on theory developed for hard real-time multiprocessor systems can be wasteful. Below, we give an example-driven explanation for why the latter holds and a high-level overview of some of the issues addressed in this dissertation. Later, in Section 1.6, a more technical explanation is provided after some needed background and concepts are set in place.

Example 1.1. Consider a real-time system composed of three sequential processes or *tasks*,[1] to be instantiated on two identical processors. Starting at time 0, each task periodically submits four units of work, also referred to as a *job*[1], once every six time units. (In other words, each task has a period of six time units and each job has an execution requirement of four time units.) Let the deadline of job k of each task, where $k \geq 1$, be at time $6k$. That is, each job is to complete execution before the next job of the same task arrives. Three possible algorithms for scheduling the tasks in this example are considered below.

Algorithm 1: In the first algorithm that we consider, one of the tasks, say Task 1, is assigned to Processor 1, and the remaining tasks to Processor 2. That is, the tasks are partitioned between the two processors, and each task runs exclusively on the processor to which it is assigned. On each processor, a pending job with the earliest deadline is executed at every instant, with ties resolved arbitrarily. A schedule for the task system based on this algorithm is shown in Figure 1.1(a). Under this algorithm, tasks do not migrate between processors, but as can be seen, Processor 2 is overloaded and jobs assigned to it miss deadlines. The amount of time by which a deadline is missed, referred to as *tardiness*, increases with time. In this example, the k^{th} job of the third task misses its deadline by $2k$ time units, for all k, and that of the second task by $2(k-2)$ time units, for $k \geq 2$. Thus, tardiness for the jobs assigned on Processor 2 grows without bound.

[1] Refer to Section 1.3.1 for formal definitions.

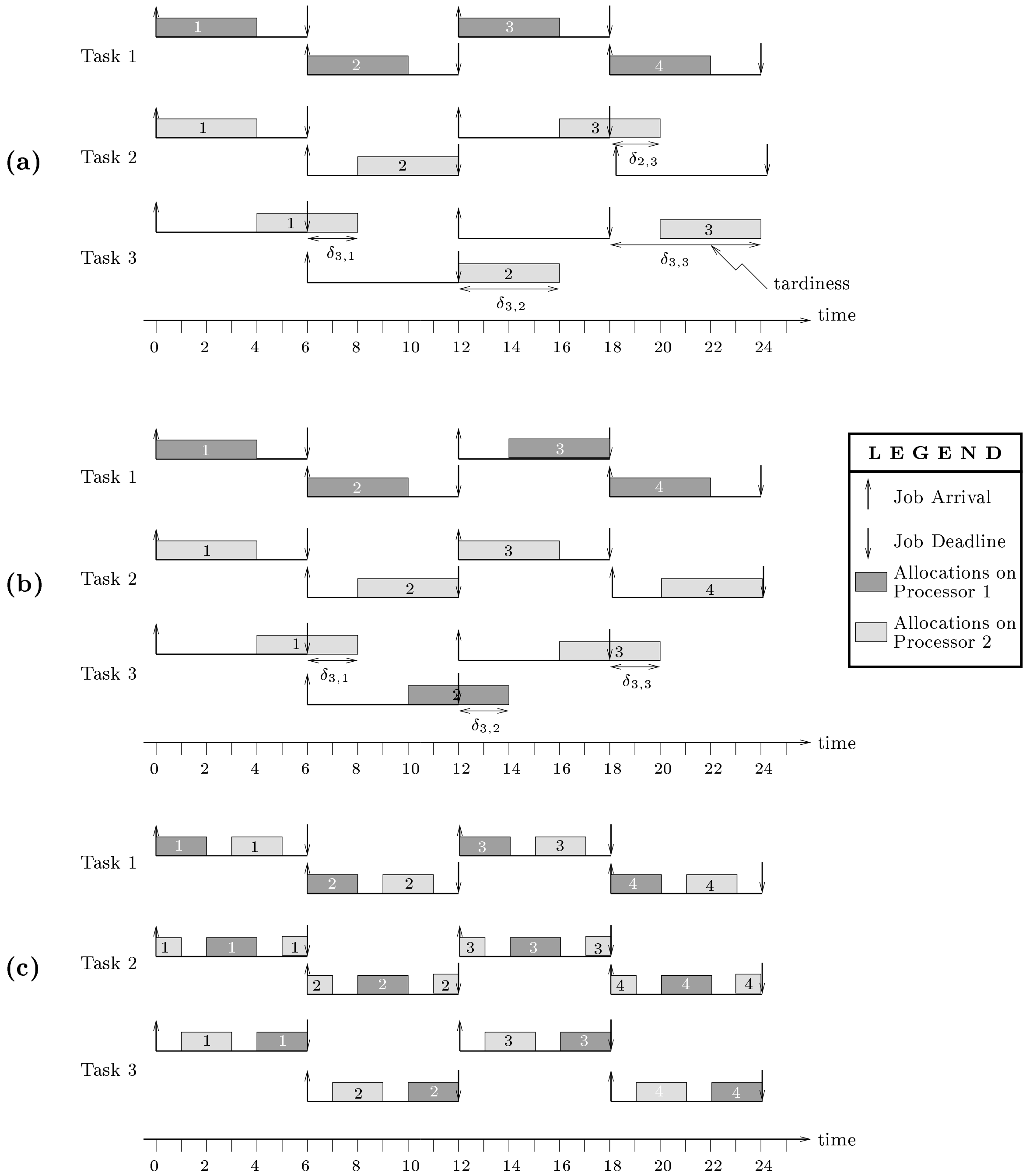


Figure 1.1: Schedules under the three algorithms described in Section 1.2.2 for the task system in Example 1.1. Numbers within rectangles indicate job numbers. If positive, the tardiness of the j^{th} job of Task i is denoted $\delta_{i,j}$.

Algorithm 2: Under this algorithm, tasks are not partitioned. At each instant, at most two pending jobs with the earliest deadlines are executed on the two processors, subject to not executing jobs of the same task concurrently. As with the previous approach, ties are resolved arbitrarily. A schedule based on this algorithm is shown in Figure 1.1(b). In this schedule, the third task migrates between the two processors, and its jobs suffer from a tardiness of two time units.

Algorithm 3. Finally, we consider scheduling using a more elaborate set of rules (the specifics of which are not of interest for the moment) that can ensure that no deadlines will be missed. A schedule under this set of rules is shown in Figure 1.1(c). Though no deadline is missed in this schedule, note that not only do tasks migrate, but every job migrates at least once. (Each job of the second task migrates twice.)

For the task system under consideration, it is easily seen that at least three processors are required to ensure that no deadlines will be missed under the first two algorithms. From Figure 1.1(b), it appears that under the second algorithm, no deadline will be missed by more than two time units. If this is the case and if bounded tardiness is tolerable, then, even though tasks may migrate under it, the second algorithm may be used to instantiate the system on just two processors, and thereby, lower the number of processors by one-third. However, real-time scheduling theory as it exists today has no tools for reliably predicting the tardiness to which arbitrary task systems may be subject under various scheduling algorithms, and in turn, lowering resource requirements, when bounded tardiness is tolerable. One of the goals of this dissertation is to extend the theory in such a direction.

Though no deadline is missed under the third algorithm, it has its own limitations, which are described in detail later in Section 1.6. One limitation is the increased number of preemptions and/or migrations for jobs, which can lead to poor cache usage, and hence, degraded performance, in practice. Furthermore, the somewhat complex scheduling model and the restrictions that the algorithm imposes can also take useful processor time away from the tasks and may in fact be overkill for soft real-time systems. For instance, one restriction of the algorithm is that job execution costs and periods be integer multiples of the quantum size, necessitating non-integral execution costs to be rounded up to the next integral value. For example, an execution cost of, say, 4.1 quanta would have to be rounded up to 5.0 quanta, which means that every fifth quantum allocated to the corresponding task would have to be partially wasted. A second goal of this dissertation is to propose techniques for scheduling when not all such restrictions can be satisfied and to determine the loss to timeliness that relaxing the restrictions entails.

With the above overview of the research addressed in this dissertation, we turn to providing needed background on real-time systems and scheduling. We begin with a description of the real-time system model assumed in this dissertation.

1.3 Real-Time System Model

To ensure that a real-time system is predictable, *a priori* knowledge of the workload of the system and available resources is necessary. A real-time task model is used to describe the workload and timing requirements of a real-time application, and a resource model is used to describe the resources that are available for instantiating the application. In this section, we first describe a commonly used hard real-time task model, upon which the soft real-time task model considered in this dissertation is based. The resource model that we assume is described afterward.

1.3.1 Hard Real-Time Task Model

In real-time terminology, a chunk of sequential work that is associated with a timing constraint (deadline) and submitted to the scheduler is referred to as a *job*. Thus, a simple model of a real-time system is a set of jobs, each of which is associated with an *arrival* or *release time*, a *deadline*, and a *worst-case execution time* (WCET). The release time of a job is the time before which the job cannot commence execution and its WCET is the maximum time that the job will execute on a dedicated processor. (If more than one processor, not all of which are identical, are part of the system's resource pool, then it is assumed that the WCET of each job is specified or can be determined for each processor.) However, enumerating all jobs is generally infeasible for most but short-lived systems, necessitating more concise models.

Periodic and sporadic tasks. As in the radar tracking system described in Section 1.1, many real-time systems consist of one or more sequential segments of code, each of which is invoked repeatedly and each of whose executions should complete within a specified amount of time. Each recurring segment of code is generally designed and implemented as a separate thread or process, and, in the terminology of real-time systems, is referred to as a *task*. Tasks can be invoked in response to events in the external environment that the system interacts with, events in other tasks, or the passage of time implemented using timers. Each invocation of a task constitutes a job, and unless otherwise specified, a task is long-lived, and can be invoked an infinite number of times, *i.e.*, can generate jobs indefinitely. Hence, many real time systems can be modeled as a set of N recurrent tasks denoted $\tau = \{\tau_1, \tau_2, \ldots, \tau_N\}$. Each task τ_i is a sequential program characterized by three parameters: a WCET, $e_i > 0$, a *minimum inter-arrival separation* or *period*, $p_i \geq e_i$, and a *relative deadline*, $D_i \geq e_i$. e_i denotes the WCET[2] for each job of τ_i, and p_i, the minimum time that should elapse between two consecutive job invocations or arrivals of τ_i. The first job can be invoked at any time. D_i denotes the amount of time within which each job of τ_i should complete execution after its release. Because τ_i is sequential, its jobs may not execute on multiple processors at the same time, *i.e.*, parallelism is forbidden. A recurrent task with the characteristics as described is referred to as a *sporadic task* and a task system composed of sporadic tasks is referred to as a *sporadic task system*. A *periodic task* is a special case of a sporadic task in which any two consecutive job arrivals are separated by exactly p_i time units, and a task system whose tasks are all periodic is referred to as a *periodic task system*. A periodic task system is said to be *synchronous* if all the tasks release their first jobs at the same time, and *asynchronous*, otherwise. All of the results in this dissertation are for sporadic task systems or their generalizations (described in Chapter 3), and hold for periodic task systems, as well. In this dissertation, we will refer to the periodic and sporadic task system classes and their generalizations collectively as *recurrent task systems*.

For a periodic or sporadic task system, the k^{th} job, where $k \geq 1$, of τ_i is denoted $\tau_{i,k}$. The release time of $\tau_{i,k}$ and its *absolute deadline* are denoted $r_{i,k}$ and $d_{i,k}$ $(= r_{i,k} + D_i)$, respectively. A job's absolute deadline is the absolute or actual time by which the job should complete execution. The qualifier absolute for the absolute deadline parameter will be omitted if unambiguous from context. If $D_i = p_i$ (resp., $D_i < p_i$) holds, then τ_i and its jobs are said to have *implicit deadlines* (resp., *constrained deadlines*). A task system in which $D_i = p_i$ (resp., $D_i \leq p_i$) holds for every task is said to be an *implicit-deadline system* (resp., *constrained-deadline system*), and one in which $D_i > p_i$ holds for one or more tasks is said to be an *arbitrary-deadline system*. Unless otherwise specified, all tasks are assumed to have implicit deadlines, and the notation $\tau_i(e_i, p_i)$ will be used to denote the parameters of τ_i concisely. A sample sporadic task system with two sporadic tasks and one periodic task is shown in Figure 1.2.

In some chapters of this dissertation, upper-case letters near the end of the alphabet, such as T, U, and V, will be used to denote tasks. In such cases, the WCET and period of a task will be denoted using the notation `TaskName`.e and `TaskName`.p, respectively, as in $T.e$ and $T.p$.

[2] As discussed later, the WCET can be suitably inflated to account for system overheads.

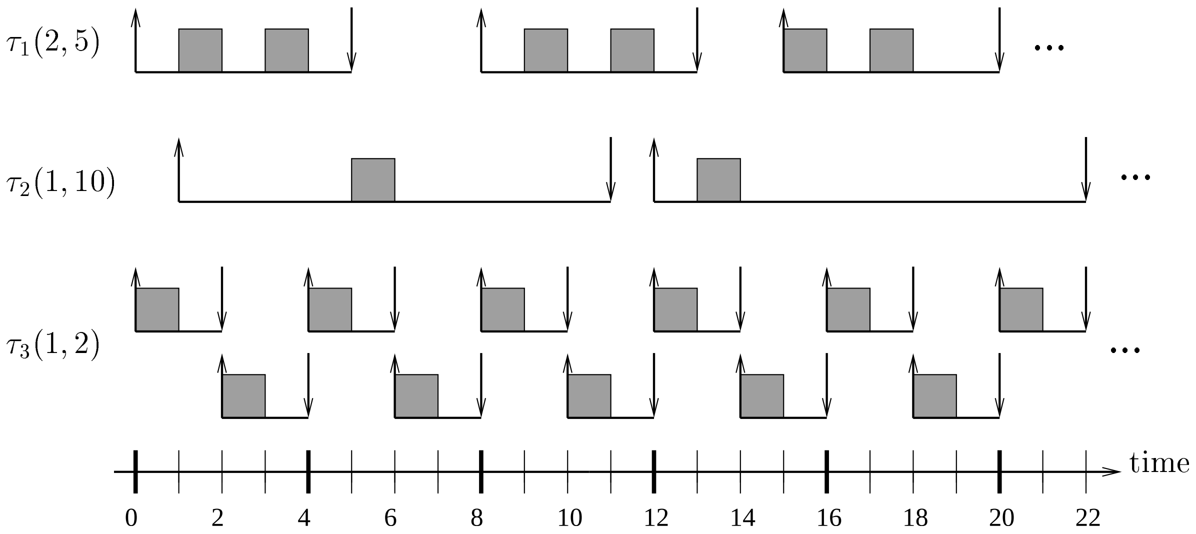


Figure 1.2: First few jobs of the tasks of an example sporadic task system scheduled on a single processor. Tasks τ_1 and τ_2 are sporadic and τ_3 is periodic. Throughout this dissertation, jobs will be depicted as in this figure: up arrows will denote job releases, down arrows, job deadlines, and shaded rectangular blocks, processor time allocations.

The ratio of the WCET to the period of a task is referred to as its *utilization*. The utilization of task τ_i is denoted $u_i \stackrel{\text{def}}{=} e_i/p_i$. With the alternative notation mentioned above, task T's utilization is denoted $T.u \stackrel{\text{def}}{=} T.e/T.p$. A task's utilization represents the fraction of a single processor that is to be devoted to the execution of its jobs in the long run. A task is said to be *heavy* if its utilization is at least 1/2, and *light*, otherwise. The maximum utilization of any task in τ is denoted $u_{\max}(\tau)$. The sum of the utilizations of all tasks in τ is referred to as the *total system utilization* of τ and is denoted $U_{sum}(\tau)$. $U_{sum}(\tau)$[3] denotes the total processing needs of τ.

Concrete and non-concrete task systems. A sporadic task system τ is said to be *concrete*, if the release time and actual execution time (which is at most the WCET) of every job of each of its tasks is specified, and *non-concrete*, otherwise. Note that infinite number of concrete task systems can be specified for every non-concrete task system. The type of the task system is specified only when necessary. Unless specified, actual job execution times are to be taken to be equal to their worst-case execution times.

[3] We will omit specifying the task system parameter in these notations if unambiguous.

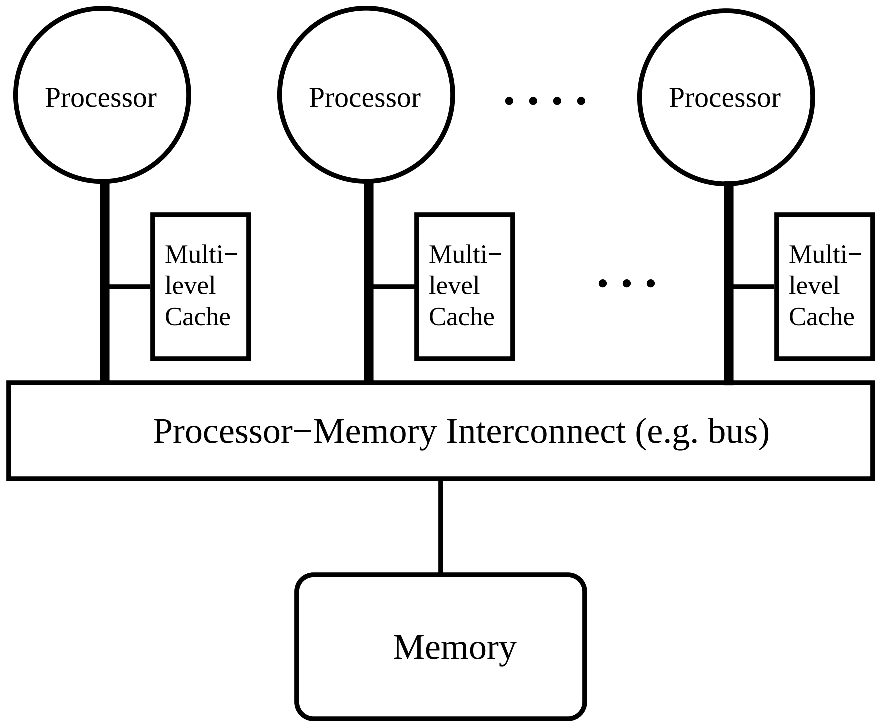


Figure 1.3: Architecture of a symmetric shared-memory multiprocessor (SMP). The processors are replicas of one another, are provided with identical caches, and have uniform access to a centralized main memory.

1.3.2 Resource Model

The focus of this dissertation is designing and analyzing algorithms for efficiently scheduling soft real-time task systems on the processors of an *identical* multiprocessor platform. Throughout this dissertation, $M \geq 2$ will denote the number of processors. As its name suggests, the processors of an identical multiprocessor are replicas of one another and have the same characteristics, including uniform access times (in the absence of contention) to memory. Uniform memory access is accomplished by the use of a centralized memory that is shared among processors. This type of multiprocessor is commonly referred to as a *symmetric shared-memory multiprocessor* (SMP). Refer to Figure 1.3 for an illustration. Each processor may be provided with one or more levels of identical caches (instruction, data, and unified) to expedite access to frequently accessed addresses and/or addresses that are spatially close. It is assumed that every task is equally capable of executing on every processor and that there is no restriction on the processors that a task may execute upon. However, the presence of caches suggests that the execution time of a job is likely to be more if the job executes on multiple processors (different processors at different times), *i.e.*, if the job *migrates*, than if it executes on a single processor. To lower migration overheads, a scheduling algorithm may choose to restrict executing a task or a job to one or a subset of the processors, even though the system model imposes no restriction. Overheads due to migration are discussed in detail in a later section. Accounting for migration and system overheads while designing a real-time system is discussed below in Section 1.3.3.

Tasks in many real-time systems require access to resources other than processors, such as memory, I/O, and network bandwidth. Similarly, tasks in many real-time systems are not completely independent. A task system is said to be *independent* if the execution of none of its tasks depends on the status of one or more of the other tasks. Some factors contributing to interdependence among tasks are synchronization constraints imposed by producer/consumer relationships [69] and a need to access to shared data structures, I/O devices, etc., in a mutually-exclusive manner, and precedence constraints, which restrict the order in which tasks may execute. In the presence of such interdependencies, tasks may block or be suspended, which will add new considerations in reasoning about resource-allocation algorithms. Integrated and holistic techniques for synergistically allocating multiple types of resources that are cognizant of synchronization and precedence constraints, and reasoning about such techniques, begin with and make use of scheduling and analytic techniques for resources of a single type. This dissertation is concerned with efficiently allocating multiple copies of the processor resource (*i.e.*, identical processors) to a soft real-time system. Allocating multiple resource types and dealing with synchronization and precedence constraints are beyond the scope of this dissertation.

1.3.3 Accounting for Overheads

Task preemptions and context switches, task migrations, and the act of scheduling itself are infrastructure or *system overheads* that are extrinsic to and take time away from the application tasks at hand. Hence, any validation approach that does not account for time lost due to overheads cannot be guaranteed to be correct. Note that no provisions are included in the task model *per se* for overheads. In a well-known method for accounting for overheads, each extrinsic activity (*e.g.*, a preemption, migration, or scheduler invocation) is charged to a unique job, and the WCET of each task is inflated by the maximum cumulative time required for all the extrinsic activities charged to any of its jobs. The extent of the overhead due to each source can vary with the scheduling algorithm, and hence, an algorithm with good properties when overheads are ignored can perform poorly in practice. Therefore, any comparison of algorithms that ignores the overheads is deficient from a practical standpoint. Throughout this dissertation, we will assume that system overheads are included in the WCETs of tasks using efficient charging methods. The WCET of a task is therefore dependent on the implementation platform, application characteristics, and the scheduling algorithm.[4]

1.4 Real-Time Scheduling Algorithms and Validation Tests

A scheduling algorithm allocates processor time to tasks, *i.e.*, determines the execution-time intervals and processors for each job while taking any restrictions, such as on concurrency, into account. In real-time systems, processor-allocation strategies are driven by the need to meet timing constraints. Before getting into a discussion of possible scheduling approaches, we define some terms and metrics commonly used in describing some properties of real-time scheduling algorithms and in comparing different algorithms.

1.4.1 Definitions

Feasibility, schedulability, and optimality. A task system τ is said to be *feasible* on a hardware platform if there is some way of scheduling and meeting all the deadlines of τ on that platform. τ is said to be *schedulable* on a hardware platform by algorithm $\mathcal{A}$, if $\mathcal{A}$ is capable of correctly scheduling τ on that platform, *i.e.*, can meet all the deadlines of τ. $\mathcal{A}$ is said to be an *optimal* scheduling algorithm if $\mathcal{A}$ can correctly schedule every feasible task system for every hardware platform. It is often useful to restrict the definition of optimality to a subset of task systems (such as periodic or sporadic task systems) or to a class of scheduling algorithms or both. When restricted to task system subsets, $\mathcal{A}$ is said to be optimal for a subset $\mathcal{S}$ of task systems, if $\mathcal{A}$ can correctly schedule every feasible task system of subset $\mathcal{S}$, and when restricted to algorithm classes,[5] $\mathcal{A}$ is said to be an optimal class-$\mathcal{C}$ scheduling algorithm, if $\mathcal{A}$ can correctly schedule every task system that can be correctly scheduled by some algorithm in class $\mathcal{C}$. Optimality can similarly be restricted to hardware platform classes, with uniprocessors and multiprocessors being the most-commonly considered classes.

[4] Alternatively, the task model can be altered to specify the WCET of a task in the absence of interferences, and explicitly list the sources of interferences and their worst-case costs. We have followed the approach that is customary in the real-time literature.

[5] Algorithm classification is discussed in Section 1.4.2.

Schedulable utilization bound. A useful and common metric for comparing different scheduling algorithms with respect to their effectiveness in meeting the deadlines of a recurrent task system is the *schedulable utilization bound*. Formally, if $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ is a schedulable utilization bound, or more concisely, *utilization bound*, for scheduling algorithm $\mathcal{A}$, then on M processors, $\mathcal{A}$ can correctly schedule every recurrent task system τ with $u_{\max}(\tau) \leq \alpha$ and $U_{sum}(\tau) \leq \mathcal{U}_{\mathcal{A}}(M, \alpha)$. If, in addition, there exists at least one task system with total utilization $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ and $u_{\max} = \alpha$, whose task parameters can be slightly modified such that U_{sum} and $u_{\max}$ are higher than $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ and α, respectively, by infinitesimal amounts, and the modified task system has a deadline miss under $\mathcal{A}$ on M processors, then $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ is said to be the *minimax utilization*[6] of $\mathcal{A}$ for M and $u_{\max}$; otherwise, $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ is a lower bound on $\mathcal{A}$'s minimax utilization for M and $u_{\max}$. The minimax utilization of $\mathcal{A}$ is also referred to as the *worst-case schedulable utilization*[7] of $\mathcal{A}$. Furthermore, if no task system with total utilization exceeding $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ can be scheduled correctly by $\mathcal{A}$ when $u_{\max} = \alpha$ on M processors, then $\mathcal{U}_{\mathcal{A}}(M, \alpha)$ is said to be the *optimal utilization bound* of $\mathcal{A}$ for M and $u_{\max}$. Note that while an optimal utilization bound is also a worst-case schedulable utilization, the converse may not hold. This is because, it is possible that there exist task systems that are correctly scheduled but have a total utilization that is higher than that of some task system that is barely schedulable (*i.e.*, some worst-case schedulable utilization). It should also be noted that when expressed using a fixed set of parameters, such as α and M, different values are not possible for the optimal and worst-case schedulable utilizations. In other words, if an optimal utilization bound exists for M and α, then a worst-case schedulable utilization that is different from the optimal bound is not possible.

Schedulability tests. In addition to serving as a comparison metric, schedulable utilizations of scheduling algorithms can also be used in devising simple and fast validation tests and online admission-control tests for the algorithms. In the context of hard real-time systems, validation tests are generally referred to as *schedulability tests*. Given the schedulable utilization $\mathcal{U}_{\mathcal{A}}(M)$ of Algorithm $\mathcal{A}$ and a task system τ, an $O(N)$-time schedulability test for τ under $\mathcal{A}$ that

[6] The phrase minimax utilization is due to Oh and Baker [93] and denotes the minimum utilization over all maximal task sets. A maximal task set is one that is schedulable but if the execution times of its tasks are increased slightly, then some deadline will be missed.

[7] Unless otherwise specified, worst-case schedulable utilizations are assumed to be with respect to M and $\alpha = u_{\max}$. It is possible to obtain other worst-case values if other or more task parameters, such as execution costs and periods, are considered.

verifies whether $U_{sum}(\tau)$ is at most $\mathcal{U}_{\mathcal{A}}(M)$, and a similar $O(1)$ per-task time online admission control test, are straightforward. The downside of such utilization-based schedulability tests is that for many algorithms, optimal schedulable utilizations are not known. In such cases, the tests are only sufficient but not necessary (*i.e.*, are not *exact* tests), and hence, can be pessimistic, *i.e.*, incorrectly conclude that deadlines may be missed.

1.4.2 Real-Time Scheduling Strategies and Classification

In general, since job release times are not known *a priori*, pre-computing schedules off-line is not possible with sporadic and some asynchronous periodic task systems. As a result, online scheduling algorithms are needed. Typically, such an algorithm assigns a priority to each job, and on an M-processor system, schedules for execution the M jobs with the highest priorities at any instant, subject to not violating constraints on migrations,[8] preemptions, concurrency, and mutually-exclusive executions, if any. We will consider scheduling[9] on uniprocessors first and that on multiprocessors afterwards.

1.4.2.1 Scheduling on Uniprocessors

In real-time systems, the need to meet timing requirements suggests using strategies optimized for that purpose. Giving higher priority to **(i)** jobs with earlier deadlines, **(ii)** those with smaller slack times,[10] or **(iii)** jobs of tasks that recur at a higher rate (*i.e.*, tasks with shorter periods) are some logically reasonable strategies for selecting jobs to execute in real-time systems. All of these are greedy strategies because each makes a choice that appears to be the best at the moment.

The algorithm that uses the first strategy is called *earliest-deadline-first* (EDF) [46]. EDF's greedy strategy turns out to be optimal for scheduling sporadic tasks on a uniprocessor [82]. Similarly, *least-laxity-first* (LLF) [91], also referred to as *smallest-slack-first*, is a scheduling algorithm, which directly uses the second strategy, and which is also optimal for sporadic task systems on a uniprocessor. Lastly, under the well-known and widely-used *rate-monotonic* (RM) scheduling algorithm, the third strategy of prioritizing tasks with shorter periods over those with longer periods is employed. Partial schedules under the three algorithms referred to are

[8]Refer to Section 1.4.2.2 for a discussion on the degree of task migrations.

[9]Unless otherwise specified, any reference to scheduling is to online approaches.

[10]The *slack* time of a job at any given time is the difference between the amount of time remaining until the job's deadline and its pending or unfulfilled execution requirement.

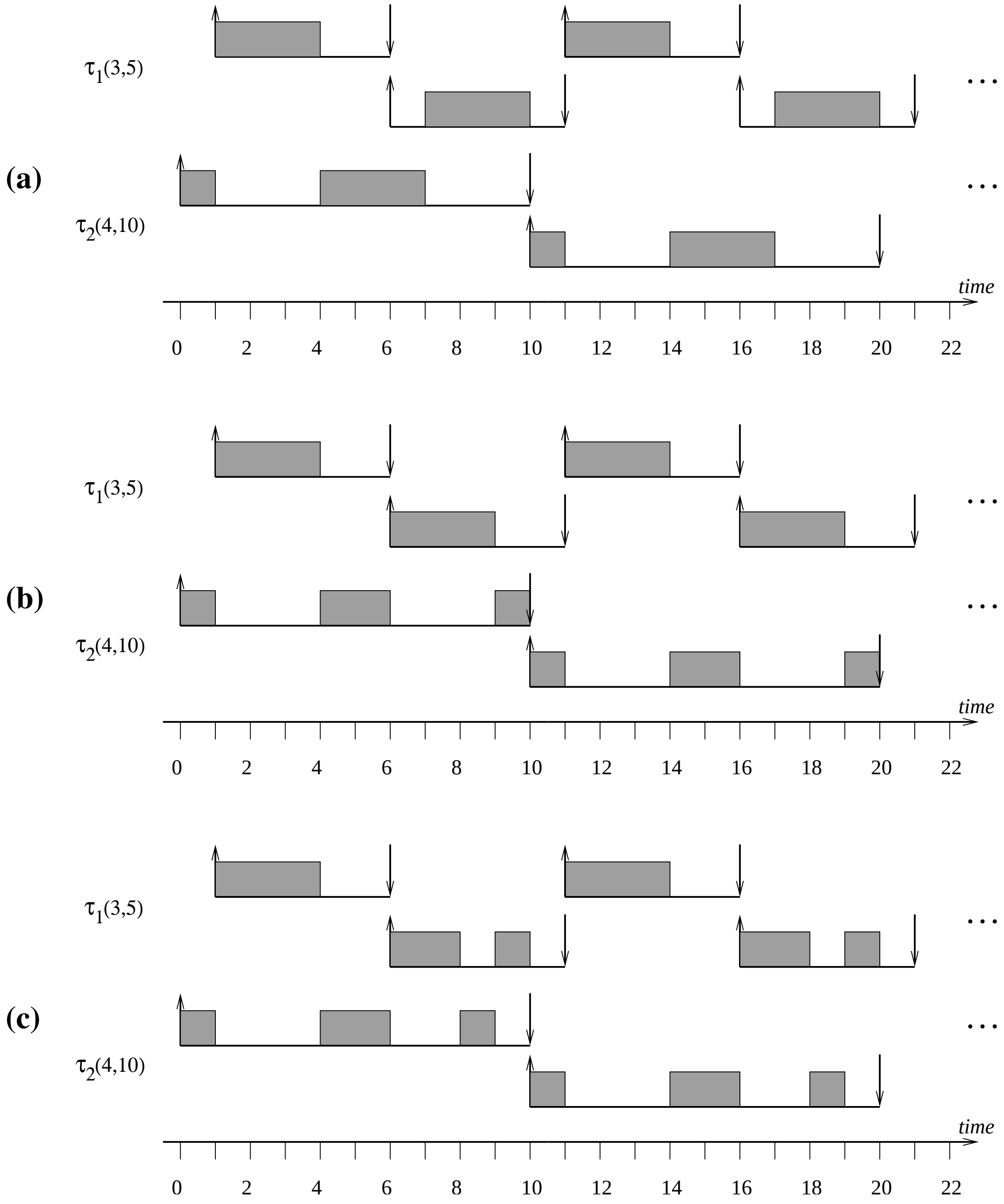


Figure 1.4: Uniprocessor schedules under **(a)** EDF, **(b)** RM, and **(c)** LLF for a task system with two tasks $\tau_1(3,5)$ and $\tau_2(4,10)$.

shown in Figure 1.4 for the first few jobs of a task system with two tasks $\tau_1(3,5)$ and $\tau_2(4,10)$. The three algorithms differ in the complexity of their priority schemes and their ability to meet the timing constraints and form the basis of a priority-based classification of scheduling algorithms presented in [42]. Before describing that classification, we briefly mention two other ways of classifying scheduling algorithms.

Preemptive and non-preemptive algorithms. Under *preemptive algorithms*, the execution of a running job can be interrupted any time before its completion and resumed later. In general, under real-time scheduling algorithms, a job is preempted only if another job with a higher priority arrives when every processor is busy. Under *non-preemptive algorithms*, a job may not be interrupted once it commences execution, and thus is guaranteed uninterrupted execution until completion. As described, preemptivity is associated with algorithms. Alternatively, preemptivity can be associated with tasks and it is often useful to consider hybrid schemes wherein whether a job can be preempted depends on whether it executes in a preemptive or non-preemptive section. Unless otherwise specified, it is to be assumed that jobs are fully preemptable.

Work-conserving and non-work-conserving algorithms. An algorithm is said to be *work conserving* if it does not idle any processor when one or more jobs are pending, and *non-work conserving*, otherwise. The most common reason for inserting idle times is to improve schedulability. For instance, under non-preemptive algorithms, idling may prevent binding a job prematurely to a processor, and hence, has the potential to correctly schedule task systems that are otherwise not schedulable. However, the time complexity associated with deciding whether idling can improve schedulability can be quite high, and hence, generally (barring a few exceptions)[10] only off-line schedulers tend to be non-work-conserving. All the algorithms considered in this dissertation, except some Pfair-related algorithms [25], are work conserving.

Priority-Based Classification

Based on how jobs and tasks are prioritized, scheduling algorithms can be classified into the following three categories.

Static-priority algorithms (P_s). Under a static-priority algorithm, each task is assigned a priority off-line, which is then assigned to each of its jobs at run-time. Thus, the priority of a task remains static, *i.e.*, unchanged, across job invocations. The RM scheduling algorithm

[10] As originally designed, Pfair scheduling algorithms [25], which can optimally schedule recurrent real-time task systems on multiprocessors, are not work-conserving. However, it has been shown that such a behavior is not necessary to improve schedulability under Pfair scheduling. This issue is discussed in more detail in Chapter 3.

mentioned above is an example of an algorithm in this class. RM is also an optimal static-priority scheduling algorithm for sporadic task systems on uniprocessors if relative deadlines are equal to periods as assumed in this dissertation.

Restricted-dynamic-priority algorithms (P_d^r). The algorithms in this class are also referred to as *task-level dynamic-priority* and *job-level fixed-priority* algorithms in the literature. Generally, under this class of algorithms, the priority of a job is determined at run-time, and hence, different jobs of a task can be assigned different priorities. However, once assigned, the priority of a job cannot be changed. EDF is an algorithm in this class.

Unrestricted-dynamic-priority algorithms (P_d^u). This third class relaxes the restriction of the previous class by allowing a given job to be assigned different priorities at different times. Hence, the relative priorities between two jobs can change with time. LLF, described earlier, is an example of an algorithm in this class.

Comparison of the Priority Classes

The restricted-dynamic-priority class generalizes the static-priority class and is in turn generalized by the unrestricted-dynamic-priority class. The effectiveness of an algorithm in meeting timing constraints can be expected to improve with increasing generalization. However, on the negative side, the implementation complexity, and run-time overheads, such as the number of context switches due to preemptions, can also increase at the same time. Below, we compare the three classes on the basis of their overheads and schedulability. Since RM, EDF, and LLF are optimal scheduling algorithms for their respective classes on uniprocessors [82, 91], we often take these three algorithms to be representatives for their classes.

Overhead comparison. Though the worst-case time complexity of selecting the highest-priority job is $O(\log N)$ (where N is the number of tasks) for each of RM, EDF, and LLF, simpler implementations that are fast in practice are possible for RM and other static-priority algorithms [84]. The maximum possible number of job preemptions, when expressed as a function of the total number of jobs, is asymptotically comparable for the static and restricted-dynamic-priority classes. However, in practice, the actual number can be higher for RM than for EDF [39]. Preemptions can be expected to be much higher for the unrestricted-dynamic-priority class and will depend on the rate at which job priorities change. For example, in

Figure 1.4, the number of preemptions under RM is twice that under EDF, and is the highest under LLF. In addition to context switches, some cache-related overheads[12] are also possible due to preemptions, and hence, it is desirable that their number be minimized.

Schedulability comparison. On a single processor, the worst-case schedulable utilization of RM for sporadic task systems is $U_{RM} = N \cdot (2^{1/N} - 1)$ [82], which is strictly decreasing with N and converges to $\ln 2 \approx 0.69$ as $N \longrightarrow \infty$. On the other hand, the schedulable utilization bound for EDF is 1.0 for all N [82]. Thus, on a uniprocessor, moving from static to restricted-dynamic-priority algorithms can significantly enhance schedulability in many cases. Furthermore, because no algorithm can correctly schedule any task system with total system utilization exceeding 1.0 on a uniprocessor, EDF is optimal not only for its class, but universally, *i.e.*, for the class encompassing all scheduling algorithms.

It should also be mentioned that the utilization-bound-based schedulability test for RM is only a sufficient test and can be pessimistic. In [77], Lehoczky *et al.* devised a more accurate but complex test for RM, which is necessary and sufficient for sporadic task systems but is only sufficient for asynchronous, periodic task systems. Using this exact characterization of RM, Lehoczky *et al.* also showed that the utilization bound of RM can be taken to be around 88% on an average. Given that RM is an optimal static-priority scheduling algorithm, there is still approximately a schedulability loss of 12% under the static-priority class on a uniprocessor.

Because EDF is universally optimal, the additional flexibility in scheduling offered by the unrestricted-dynamic-priority class (LLF, which belongs to the class, is also universally optimal) provides no extra benefit at least for independent task systems. Further, since their run-time overheads tend to be higher, unrestricted-dynamic-priority algorithms are not considered interesting on uniprocessors.

1.4.2.2 Scheduling on Multiprocessors

In this subsection, we consider multiprocessor scheduling in some detail.

Multiprocessor Scheduling Approaches

Two approaches traditionally considered for scheduling on multiprocessors are *partitioning* and *global scheduling*.

[12]Cache-related overheads due to preemptions and migrations are discussed further towards the end of Section 1.4.2.2.

Partitioning. Under partitioning, the set of tasks is statically partitioned among processors, that is, each task is assigned to a unique processor upon which all its jobs execute. Each processor is associated with a separate instance of a uniprocessor scheduler for scheduling the tasks assigned to it and a separate local ready queue for storing its ready jobs. In other words, the priority space associated with each processor is local to it. The different per-processor schedulers may all be based on the same scheduling algorithm or use different ones. The algorithm that partitions the tasks among processors should ensure that for each processor, the sum of the utilizations of tasks assigned to it is at most the utilization bound of its scheduler. Scheduling under this model is depicted in Figure 1.5(a).

Global scheduling. In contrast to partitioning, under global scheduling, a single, system-wide, priority space is used, and a global ready queue is used for storing ready jobs. At any instant, at most M ready jobs with the highest priority (in the global priority space) execute on the M processors. No restrictions are imposed on where a task may execute; not only can different jobs of a task execute on different processors, but a given job can execute on different processors at different times, subject to not violating restrictions on non-preemptivity. Figure 1.5(b) illustrates global scheduling.

Two-level hybrid scheduling. Some algorithms do not strictly fall under either of the above two categories, but have elements of both. For example, algorithms for scheduling systems in which some tasks cannot migrate and have to be bound to a particular processor, while others can migrate, follow a mixed strategy. In general, scheduling under a mixed strategy is at two levels: at the first level, a single, global scheduler determines the processor that each job should be assigned to using global rules, while at the second level, the jobs assigned to individual processors are scheduled by per-processor schedulers using local priorities. Several variants of this general model and other types of hybrid scheduling are also possible. Typically, the global scheduler is associated with a global queue of ready, but unassigned, jobs, and the per-processor schedulers with queues that are local to them. Elements of one type of hybrid scheduling are illustrated in Figure 1.5(c).

Migration-Based Classification

As can be seen, the extent to which tasks migrate across processors differs under the three scheduling approaches discussed above. Hence, apart from a priority-based classification de-

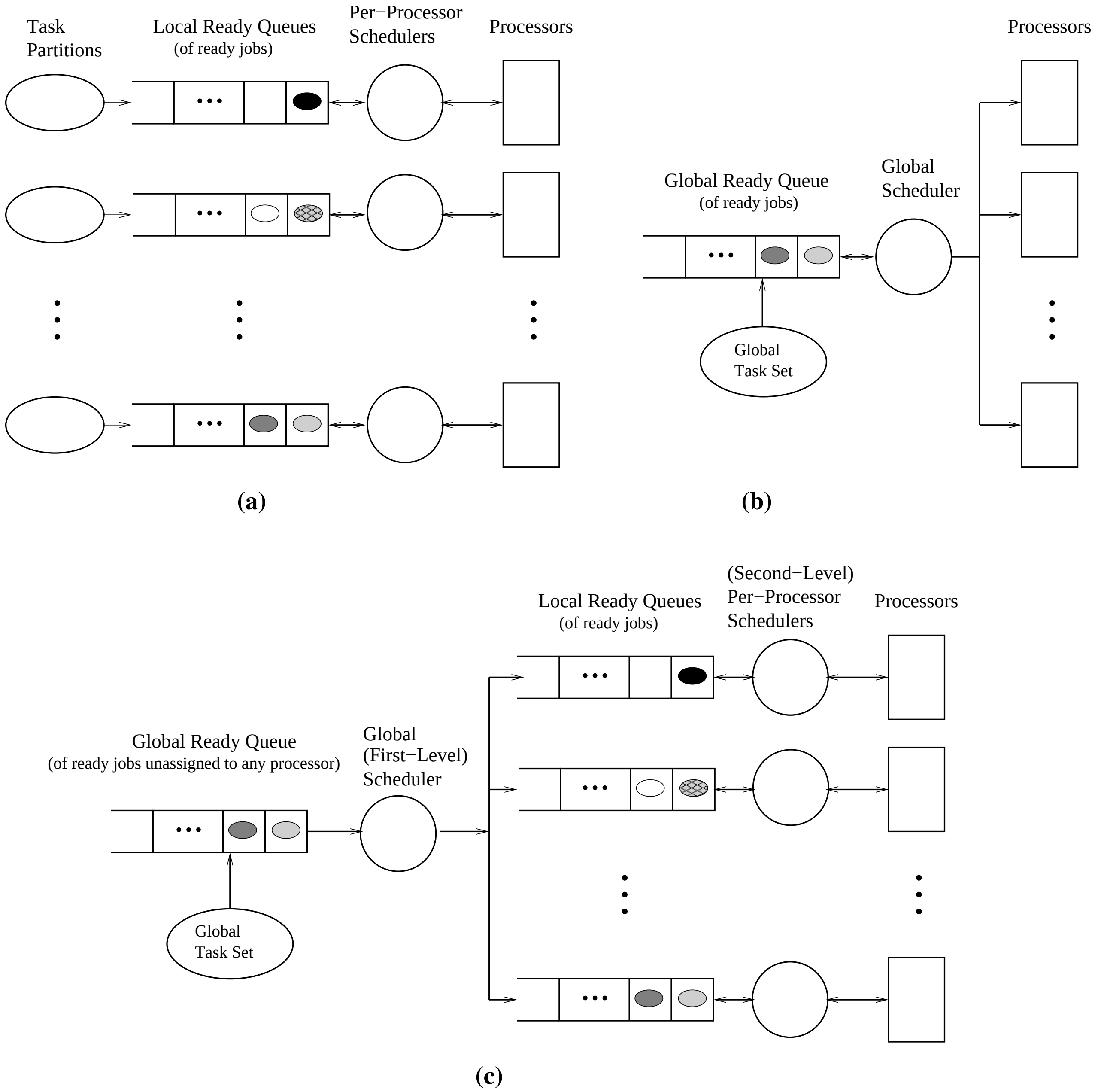


Figure 1.5: Schematic representations of **(a)** partitioned, **(b)** global, and **(c)** two-level hybrid scheduling algorithms.

scribed in Section 1.4.2.1, multiprocessor scheduling algorithms can be classified based on the degree of migration they allow to tasks as well, as suggested by Carpenter *et al.* in [42]. We next describe their classification.

No-migration or partitioned algorithms (M_p)**.** Under this class of algorithms, task migration is prohibited. Hence, the algorithms in this class use the partitioning strategy described above. Partitioned-EDF, in which each per-processor scheduler is EDF, is an example of an algorithm in this class.

Restricted-migration algorithms (M_r)**.** This class of algorithms allows tasks to migrate at job boundaries. That is, different jobs of a task can execute on different processors, but a given job may not migrate and must execute only on a single processor. Some algorithms in this class may require two-level, hybrid scheduling as described above. Restricted-migration EDF, referred to as r-EDF and described in [24], is an example of an algorithm in this class that requires two-level scheduling. Under r-EDF, a global scheduler assigns each newly-arriving job to a processor that can accommodate it, *i.e.*, can correctly schedule all the jobs previously assigned to it as well as the new job. The jobs assigned to each processor are scheduled locally by a second-level EDF scheduler. Second-level scheduling is not required for all the algorithms in this class and can be eliminated if scheduling rules permit it. Global, non-preemptive EDF (referred to as g-NP-EDF), with no restrictions on where different jobs execute, is an algorithm in this class for which single-level scheduling suffices.

Full-migration algorithms (M_f)**.** A full-migration algorithm places no restriction on inter-processor migration and uses the global scheduling approach. Global, preemptive EDF (referred to as g-EDF) is an example of an algorithm in this class.

Carpenter *et al.* suggested considering each migration class with each priority class, and hence, nine classes of multiprocessor scheduling algorithms. Their taxonomy of scheduling algorithms is shown in the form of a matrix in Table 1.1. When convenient, we will denote a class using a tuple (x, y), where x denotes a priority class (one of P_s, P_d^r, and P_d^u), and y, a migration class (one of M_p, M_r, and M_f). The entries in the table (discussed later) represent known upper and lower bounds on the worst-case schedulable utilization for each class.

Comparison of Classes of Multiprocessor Schedulers

Schedulability comparison. In [42], Carpenter *et al.* compare the schedulability of the classes with respect to two aspects: **(i)** the set of all task systems that any algorithm in each class can schedule correctly (which we refer to as *raw schedulability*) and **(ii)** the best known schedulable utilization bounds for any algorithm in each class. Their results are summarized

M_f: **full migration**	$\frac{M^2}{3M-2} \le U \le \frac{M+1}{2}$ [16]	$U = M - \alpha(M-1)$, if $\alpha \le \frac{1}{2}$ [61] $U = \frac{M+1}{2}$, otherwise [22]	$U = M$ [25, 105]
M_r: **restricted migration**	$U \le \frac{M+1}{2}$	$U \ge M - \alpha(M-1)$, if $\alpha \le \frac{1}{2}$ [24] $U = \frac{M+1}{2}$, otherwise [24]	$U \ge M - \alpha(M-1)$, if $\alpha \le \frac{1}{2}$ $U = \frac{M+1}{2}$, otherwise
M_p: **partitioned**	$U = \frac{M+1}{2}$ [17]	$U = \frac{\beta M+1}{\beta+1}$, where $\beta = \left\lfloor \frac{1}{\alpha} \right\rfloor$ [86]	$U = \frac{\beta M+1}{\beta+1}$, where $\beta = \left\lfloor \frac{1}{\alpha} \right\rfloor$
	P_s: **static**	P_d^r: **restricted dynamic**	P_d^u: **unrestricted dynamic**

Table 1.1: Carpenter *et al.*'s classification of multiprocessor scheduling algorithms presented in [42]. Entries in the table represent known lower and upper bounds on the worst-case schedulable utilization, U, for the different classes of scheduling algorithms. (Some entries have been updated to reflect later advances.) $\alpha = u_{\max}$, the maximum utilization of any task in the task system under consideration. Citations next to the entries indicate their primary sources.

below.

Comparison of raw schedulability. With respect to raw schedulability, Class C_1 is said to be as powerful as (resp., strictly more powerful than) Class C_2, if the set of all task systems correctly scheduled by any algorithm in C_2 is a subset of or the same as (resp., proper subset of) the set of all task systems correctly scheduled by some algorithm in C_1. Some salient raw-schedulability comparisons are as follows.

Comparison within a migration class: *Within a migration class, increased generalization along the priority dimension does not worsen raw schedulability.* Specifically, the P_d^r class is strictly more powerful than the P_s class when both are subject to the same migration rules. The two dynamic-priority classes are equally powerful under partitioning and the P_d^u class is at least as powerful as (resp., strictly more powerful than) the P_d^r class within the M_r class (resp., M_f) class.

Comparison within a priority class: It is natural to expect a relationship analogous to the one above to exist among the migration sub-classes of a priority class, wherein M_f is the most powerful, and M_p, the least. However, in contrast to the above, with the exception of the (P_d^u, M_f) (top, right) class, *migration classes within a priority class are not guaranteed to remain as powerful or improve with increasing generalization.* For instance, within the P_d^r class, the M_f sub-class (*i.e.*, the (P_d^r, M_f) class) is not comparable to the M_p sub-class (*i.e.*, the (P_d^r, M_p) class), that is, there exist instances of sporadic task systems that can be correctly scheduled by the (P_d^r, M_f) class but not by the (P_d^r, M_p) class, and vice versa.

Comparison of priority classes across migration classes: With the exception of comparisons involving the (P_d^u, M_f) class, *no relationship can be established among priority classes if they are not all subject to the same migration rules*. For instance, though the (P_d^r, M_f) class is more powerful than the (P_s, M_f) class, the (P_d^r, M_f) class is incomparable to the (P_s, M_p) class.

In a nutshell, the (P_d^u, M_f) class is the most powerful; however, the remaining classes cannot be compared if they do not fall under the same migration class. Further, it is evident from the above comparison that on multiprocessors, EDF, which belongs to the P_d^r class, is not universally optimal, regardless of the degree of migration allowed. Finally, though not implied by the above discussion, it turns out that LLF, which falls under the P_d^u class, is also not universally optimal, even if migrations are unrestricted, which is somewhat surprising.

Based on examples provided by Carpenter *et al.* in [42], the observed incomparability in raw schedulability when priorities are restricted can be explained in part as follows. We first explain why a full-migration algorithm may fail to correctly schedule a task system that can be so scheduled by a partitioned or a restricted-migration algorithm. Recall that tasks are sequential and may not execute concurrently. Hence, a key to effective global scheduling on multiprocessors lies in minimizing, if not eliminating, intervals during which there are more processors than pending tasks, forcing processors to idle. However, idling cannot effectively be minimized in all cases in task systems in which tasks or jobs are not pinned to processors by the obvious greedy strategies or schemes with restrictions on priority assignments. This is illustrated in Figure 1.6. The task system in this figure (which is provided in [42]) has a total utilization of 2.0 and can be scheduled correctly on two processors if partitioned. Because this task system fully loads two processors, idle time in a processor will be followed by a deadline miss, at least when job releases are periodic. Under partitioning, tasks τ_2 and τ_4 are assigned to different processors and hence are in different priority spaces. Therefore, there are no rules that govern how τ_2 and τ_4 execute with respect to one another. As can be seen from inset (a), the execution of the first jobs of τ_2 and τ_4 are somewhat interleaved. However, such interleaved execution of any two jobs (which can avoid processor idling) is impossible

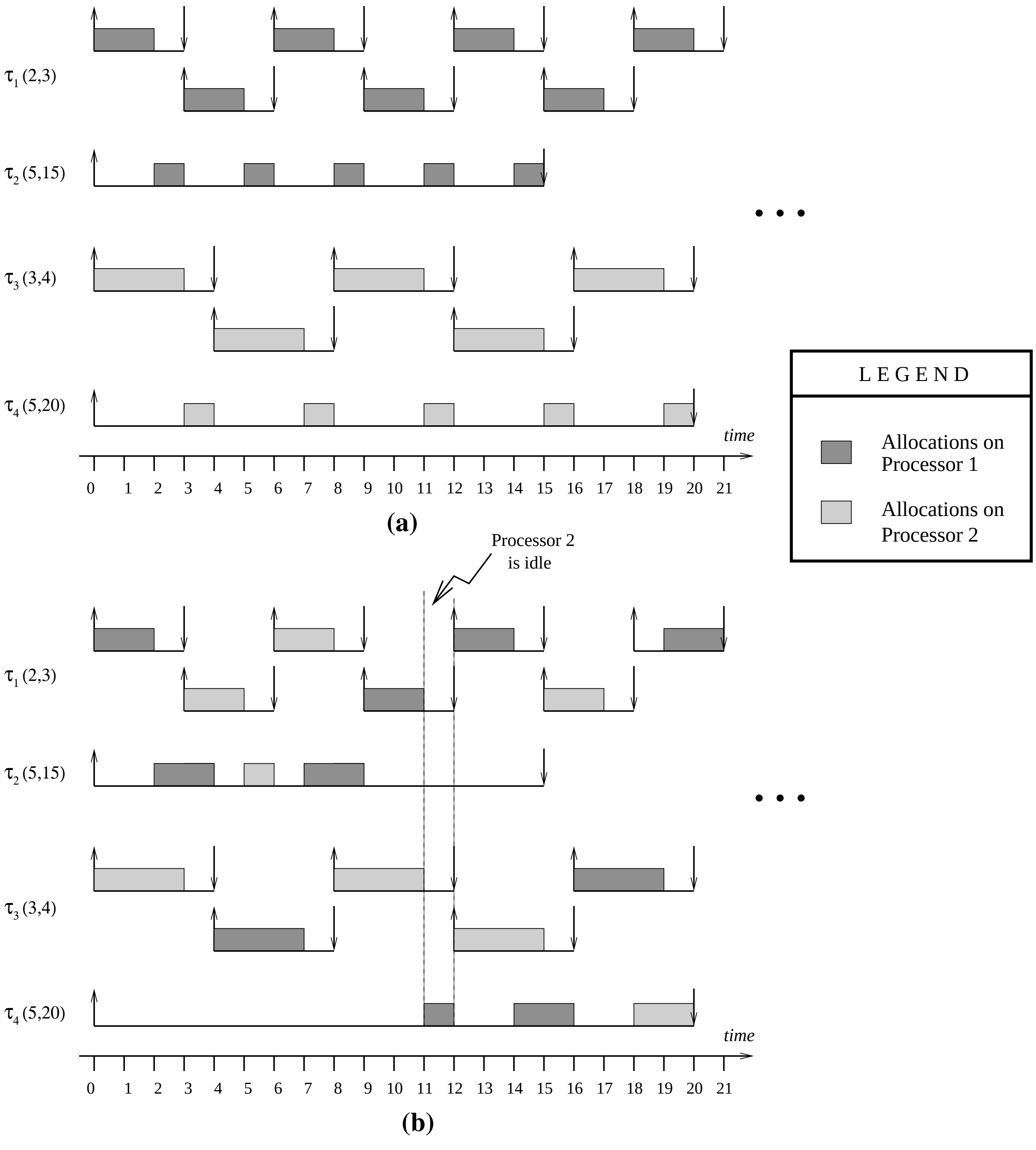


Figure 1.6: Partial schedules under **(a)** partitioned-EDF and **(b)** g-EDF for four tasks on two processors.

under a full-migration algorithm with restrictions on priority assignments to jobs (*i.e.*, one in which job priorities cannot be changed once assigned). This is because, the execution of two jobs belonging to the same priority space can be interleaved only if each has a higher priority than the other at some point in time. Carpenter *et al.* have found that no algorithm in the

(P_d^r, M_f) class can schedule this task system without deadline misses [42]. As an example, we have provided a schedule under g-EDF in inset (b). In this schedule, Processor 2 is idle from time 11 to time 12, implying that a deadline will be missed sometime in the future (at least if the task system is periodic because $U_{sum} = 2.0$). The schedule in inset (a) is also a schedule under restricted-migration EDF (r-EDF), and a similar reasoning as above can be used to explain why an algorithm in the M_r class may be capable of scheduling a task system that no algorithm in the M_f class can.

We next explain why partitioning may sometimes schedule task systems that cannot be scheduled by algorithms in the M_r class. Recall that the algorithms considered are work conserving. Hence, no processor may be idled when a ready job is waiting under full-migration policies, or when an unassigned ready job exists under restricted-migration policies. While scheduling in a work-conserving manner is not problematic in the case of full-migration algorithms, under restricted-migration algorithms, binding a job to a processor as soon as a processor becomes available can prove to be premature in some cases if the system is not under full load, *i.e.*, $U_{sum} < M$. Example schedules for a task system (this task system is also from [42]) under partitioned-EDF and two algorithms in the (P_d^r, M_r) class are shown in Figure 1.7. As can be seen, under partitioning, the second job of τ_2 is prohibited from executing on Processor 2 between time 6 and time 7, even though it is idle. This in turn, helps in preventing a deadline miss.

Due to the above two reasons (namely, an inability to interleave the execution of different jobs, and an inability to postpone binding a job to a processor for the requisite amount of time), when there are restrictions on how priorities may be assigned or on job migrations, there exist task systems that can be scheduled correctly only if partitioned. Similarly, there exist task systems that cannot be partitioned among processors without exceeding the schedulable utilization of at least one processor, but which can be correctly scheduled otherwise (despite restricting priorities or not allowing full migration), and hence the incomparability.

Comparison of schedulable utilization bounds. The best known lower and upper bounds on the schedulable utilization for any algorithm in each of the nine classes are provided in Table 1.1. The top, left entry in the table means that there exists some algorithm in the full-migration, static-priority class that can correctly schedule every task system with total utilization at most $\frac{M^2}{3M-2}$, and that there exists some task set with total utilization slightly higher than $\frac{M+1}{2}$ that cannot be correctly scheduled by any algorithm in the same class. The

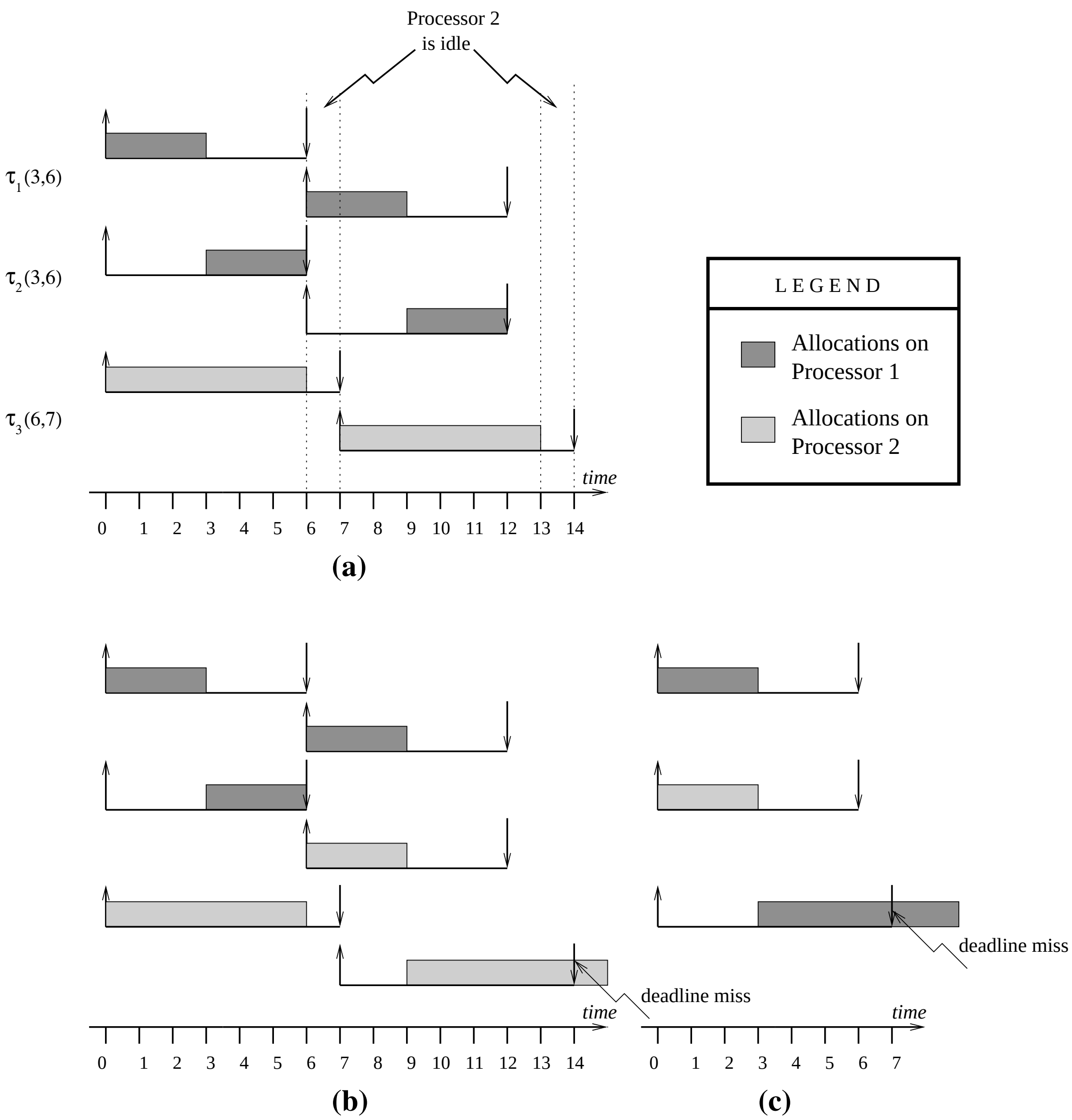


Figure 1.7: Partial schedules under **(a)** partitioned-EDF and **(b)** & **(c)** two algorithms in the (P_d^r, M_r) class for three tasks on two processors.

other entries in the table have a similar interpretation. An "equals" operator in an entry indicates that the upper and lower bounds match, and hence, the corresponding value represents the worst-case schedulable utilization.

As can be seen from the table, the worst-case schedulable utilization is not yet known for the two non-partitioning-based P_s classes, and the known lower bound is poorer for the M_f class than for the M_p class. A similar relationship exists within the P_d^r class also if $u_{\max}$ is taken into account. On the other hand, if no restriction is imposed on $u_{\max}$, then both the M_f and

M_p classes within the P_d^r class have identical worst-case schedulable utilizations. In general, it appears that all the three migration sub-classes have comparable schedulable utilizations when taken within either the static- or restricted-dynamic-priority classes. Further, as can be seen from the table, the schedulable utilizations of these classes cannot exceed $\frac{M+1}{2}$ (which is roughly 50%) in the worst case. However, in sharp contrast to these two priority classes, easing the restrictions on migrations can increase schedulability to 100% under the unrestricted dynamic-priority class.

Comparison of preemption and migration overheads. Unlike schedulability, which, contrary to normal expectations, does not always improve as migrations are more liberally allowed, run-time overhead generally increases as the migration rules become more general. Some significant components of the overhead imposed by migration are discussed below. But first, a general description of the memory hierarchy present in computers is in order.

Modern computers are provided with a memory hierarchy to speed up accesses to memory references that exhibit spatial or temporal locality. Registers and different levels of caches are above the main memory in this hierarchy and are local to processors. Access times are longer to storage elements lower down in the hierarchy, and since an address is likely to be available at different levels at different times, not all accesses to a given address will have the same access time at all times or from all processors. The differences among the access times to registers and different cache levels are generally negligible compared to the difference between the access times to the lowest-level cache and the main memory. Hence, we will let the term cache refer collectively to all storage above main memory.

The presence of a memory hierarchy immediately suggests that a job will complete executing sooner if its memory requests can be served from the cache. Spatial and temporal locality exhibited by the memory-access requests of most programs suggests that it is more likely for such requests of a job to be served from the cache, if the job executes on a single processor and without preemptions. Thus, a major overhead due to migrations is that due to a loss of cache affinity, *i.e.*, the time needed to fetch the contents of addresses that would have otherwise been serviced in-cache, from main memory. With *write-back*[13] caches, this overhead includes the time needed to invalidate associated cache lines, if needed, on the first processor (*i.e.*, the

[13]Caches generally use one of the two policies for updating memory: *write-through* and *write-back*. In write-through caches, any write to an address is to both the cache and the memory, whereas in write-back caches, a write to memory is deferred until the associated cache line is replaced or the contents are needed by some other processor.

processor that the job executed upon prior to preemption) apart from the time needed to fetch the contents from that processor's cache or the main memory. Another significant component of the migration overhead is the time needed to invalidate the cache lines containing a job's process control block (PCB) from the first processor's cache (if it is dirty), refetch it into the cache of the second processor, and load needed process status registers. On modern processors with support for virtual memory, recently-used page table entries of a process may also have to be invalidated and refetched.

Note that some of the overheads described above can be incurred even if a preempted job resumes execution at a later time on the same processor, that is, even if a job is only preempted but does not migrate. The amount of overhead will depend on whether its cache lines were evicted by jobs that executed in the intervening time. Since the state of the cache cannot be predicted reliably, it is common to assume a cold cache as the worst case even for preemptions. However, in practice, cache-related overheads are likely to be more pronounced if a job really migrates, and hence overhead due to migration also increases with increased generalization.

The crux of the comparison of the priority and migration classes presented above and in Section 1.4.2.1 can be stated as follows. *Increased generalizations along the priority and migration dimensions come at a price that may be acceptable if the **gain** in schedulability that the generalization enables outweighs the **loss** it imposes due to overheads.*

The loss incurred as we move from static-priority schedulers to restricted-dynamic-priority schedulers due to the overheads of more complex scheduling should be negligible on modern systems (and perhaps even on systems that are less powerful) in comparison to the increase in schedulability. As discussed in Section 1.4.2.1, for some task systems, moving to a restricted-dynamic-priority scheduler can in fact lower the number of preemptions, leading to an overall decrease in the loss due to overheads. Hence, in this dissertation, we consider only the two dynamic-priority categories. We close this section by summarizing the trade-offs between flexibility and overheads offered by increasing migration within dynamic-priority categories.

1.4.2.3 Overheads versus Flexibility Trade-offs

Though migrations and preemptions can add to the overheads incurred, allowing them improves a scheduler's flexibility, and can help in meeting more timing constraints in real-time systems. Recall that the full-migration, unrestricted-dynamic-priority class is more powerful than every other class. Furthermore, if migrations are prohibited, then tasks have to be partitioned among the available processors. Though cache-related overheads are the least with partitioning, this method suffers from two significant drawbacks. First, task partitioning should ensure that no processor's capacity is exceeded, and so is a bin-packing problem,[14] which is NP-hard in the strong sense [59]. Hence, determining an optimal task assignment is intractable except for very small task systems, and sub-optimal heuristics are commonly used. Second, partitioning may be a cumbersome strategy in *dynamic*[15] task systems, in which task composition, *i.e.*, the set of tasks and/or their utilizations can change at run-time, invalidating the current partitioning and necessitating a repartitioning. The worst-case time complexity of commonly-used partitioning heuristics is at least $O(NM)$, and hence, partitioning can be prohibitive to perform at run-time. Furthermore, if repartitioned, tasks have to migrate, and the prime benefit of partitioning, namely, lack of migrations and processor affinity for tasks, is lost.

Apart from the above drawbacks of partitioning, recent trends and developments in processor and computer architecture have led to designs that can considerably mitigate migration overheads, and hence, augur well for global scheduling algorithms. One significant trend is the emergence of multicore architectures, which have multiple processing units on a single die. It is widely believed that multicore architectures are the way to achieve higher processor performance, given that thermal and power problems impose limits on the performance that a single processing unit can deliver. In some multicore-based designs, the different processor cores share one or more levels of on- or off-chip caches. Hence, tasks do not significantly lose cache affinity when they migrate, and so, overheads due to migration are rendered almost negligible.

Thus, as things currently stand, no class is clearly superior than the rest, and it can be expected that the choice of the algorithm depends on the characteristics of the underlying platform and the application at hand. However, as shown in Figure 1.1(a), pure partitioning has no scope of improving system utilization if a task system cannot be partitioned among the available processors, even if bounded tardiness is tolerable. Hence, since our focus in this dissertation is soft real-time systems, we do not consider no-migration algorithms. As already discussed, we do not consider static-priority algorithms either. Thus, our focus is restricted to the upper, right (2×2) sub-matrix of Table 1.1.

---

[14] Here, processors are the bins, tasks are items to be packed, schedulable utilizations of the processors' scheduling algorithms are the bin sizes, and task utilizations are item sizes.

[15] There is no relation between dynamic task systems and dynamic priorities. A dynamic task system can be scheduled by a static-priority scheduling algorithm.

With the above introduction to real-time systems and associated concepts, and hard real-time scheduling on multiprocessors in place, we are ready to describe soft real-time systems next.

1.5 Soft Real-Time Systems

As mentioned in Section 1.1, violations of soft real-time constraints are not disastrous and can be tolerated occasionally or by bounded amounts. A task with soft (resp., hard) real-time constraints is said to be a *soft* (resp., *hard*) *real-time task*. A real-time system may either consist exclusively of either hard real-time tasks or soft real-time tasks only, or be a combination of the two. As a simple example, consider a radar tracking system tracking both hostile and friendly targets. In such a system, deadlines of tasks that track hostile targets must not be missed, whereas those of tasks that track friendly targets may be missed by bounded amounts.

It should be noted that the definition of soft real-time is somewhat broad and can span a wide spectrum of systems. The nature and extent of violations of timing constraints permissible for soft real-time tasks is application dependent. Determining permissible levels of violation is beyond the scope of this dissertation and it is assumed that such levels are provided. Some significant soft real-time models proposed in the literature that differ in the nature of permissible violations are reviewed in Chapter 2. Below, we merely describe the model considered in this dissertation.

Soft real-time task model. The soft real-time task model that we consider augments the sporadic and periodic task models described in Section 1.3 by associating a tardiness threshold with every task in the task system. If δ denotes the tardiness threshold of a soft real-time task, then any job of the task can be late, *i.e.*, miss its deadline, by at most δ time units. Though there exist task systems composed of tasks with different tolerances to tardiness, in this dissertation we restrict our attention primarily to systems whose tasks have equal tolerance to tardiness.

Formally, the *tardiness* of a job $T_{i,j}$ in schedule $\mathcal{S}$ is defined as $\mathit{tardiness}(T_{i,j}, \mathcal{S}) = \max(0, t - d_{i,j})$, where t is the time at which $T_{i,j}$ completes executing in $\mathcal{S}$. The tardiness of a task system τ under scheduling algorithm $\mathcal{A}$, denoted $\mathit{tardiness}(\tau, \mathcal{A})$, is defined as the maximum tardiness of any job of any task in any schedule for any concrete instantiation of τ under $\mathcal{A}$. If under

$\mathcal{A}$, κ is the maximum tardiness on M processors for any task system with $U_{sum} \leq M$, then $\mathcal{A}$ is said to *ensure a tardiness bound of* κ on M processors. Though tasks in a soft real-time system are allowed to have non-zero tardiness, we assume that *missed deadlines do not delay future job releases*. That is, even if a job of a task misses its deadline, the release time of the next job of that task remains unaltered. For example, in the schedules in Figures 1.1(a) and 1.1(b), though jobs of Task 3 miss their deadlines, releases of future jobs are not impacted. Hence, guaranteeing a reasonable bound on tardiness that is independent of time is sufficient to ensure that in the long run each task is allocated a processor share that is in accordance with its utilization. Because each task is sequential and jobs have an implicit precedence relationship, a later job cannot commence execution until all prior jobs of the same task have completed execution. Thus, a missed deadline effectively reduces the interval over which the next job should be scheduled in order to meet its deadline or not exceed its tardiness threshold.

Model applicability. Digital-signal-processing (DSP) systems for tracking users in virtual-reality (VR) applications and multimedia systems are some examples of soft real-time systems for which the model under consideration may be appropriate. Each of these systems can be modeled as a periodic or a sporadic task system in which it is desirable that each job of each task completes execution before the next job of the task is released. However, a small bounded tardiness for any job is also generally tolerable, especially if such tolerance can lead to a better utilization of system resources or improved flexibility.

As a specific example, consider a video decoding task that is decoding a video stream at the rate of 30 frames per second in a multimedia system. While it is desirable that the task decode every frame within 33.3 ms, a tardiness of the order of a few milliseconds will not compromise the quality of output if the rate of decoding is still 30 frames per second over reasonably long intervals of time. Tardiness may add to jitter in job completion times, but it is unlikely that jitter of the order of a few tens of milliseconds will be perceptible to the human eye. Similarly, tardiness will add to the buffering needs of a task, but should be reasonable, if the maximum tardiness is reasonably bounded, and a system designer is able to choose a tardiness value that balances the processing and memory needs of the system.

It may be argued that a real-time model in which no job misses its deadline, but where jobs can occasionally be skipped, may be better suited for the video decoding example.[16] However,

[16]This model belongs to a class of models referred to as *firm real-time* models, which assume that there is *no* value to completing a task late. The firm real-time model is discussed in Chapter 2.

this model may have the disadvantage that optimally determining whether to skip a frame can incur additional scheduling overhead at run-time. This is because not too many frames can be skipped from the same stream and every frame that is not skipped should meet its deadline. Further, to our knowledge, no experimental evidence is available as to which is a better approach, and it appears that if tardiness is minimal, then it may actually be preferable to not skip frames. Also, if the video stream is MPEG-coded, then it may not be possible to skip frames arbitrarily. Specifically, since the I and P frames are needed in decoding B frames and other P frames, there is less freedom in choosing frames to skip. In the case of DSP systems, since the accuracy of the computational results on a later sample increases if the results of processing the most recent sample are available, there is value to completing tasks as long as they are not too late.

A final argument in support of the model of interest to us is as follows. As discussed in Chapter 2, almost all the work on the other soft real-time models considered in the literature is with respect to uniprocessors and is concerned with dealing with overload, *i.e.*, scenarios in which the total system utilization exceeds the available utilization, which is 1.0. Overload could be over long terms, such as system lifetime, or occasional and for short durations, as in systems whose average loads do not exceed the available capacity but whose instantaneous loads could exhibit significant variations. However, in the context of multiprocessors, there exist scheduling algorithms under which guarantees on timeliness that can be provided are not known when the total utilization exceeds the schedulable utilization (*i.e.*, $\mathcal{U}(M)$) of the algorithm, but is much lower than the total available capacity (*i.e.*, M), *i.e.*, even when the system is much below full load. This is true of all the algorithms in Table 1.1, except the optimal algorithms, which belong to the (P_d^u, M_f) class, necessitating a study of the model. We conjecture that results based on the model considered may, in fact, be necessary in dealing with overload.

With needed background and concepts in place, we return to the subject of this dissertation, namely, facilitating cost-effective implementations of soft real-time systems on multiprocessors. We begin by articulating a need for research in this direction.

1.6 Limitations of State-of-the-Art

As noted in Section 1.2.2, the current multiprocessor-based real-time scheduling theory is too expensive for direct application to soft real-time systems. Below, we briefly explain why this is the case.

Deficiencies of non-optimal algorithms. A necessary and sufficient condition for the feasibility of a recurrent task system τ on M processors is that $U_{sum}(\tau) \leq M$ hold [25, 14]. (As explained in Section 1.3.3, U_{sum}[17] denotes the total system utilization after accounting for all the run-time overheads including scheduling, context-switching, preemption, and migration overheads.) From the discussion in Section 1.4.2.2 of the entries in Table 1.1, it is known that algorithms in eight of the nine classes (*i.e.*, all the classes except the (P_d^r, M_f) class) are not optimal, and require restrictions on total system utilization that can approach roughly 50% of the available processing capacity if every deadline needs to be met. Hence, when bounded tardiness is tolerable, using the utilization bounds given in Table 1.1 for validation with algorithms in those classes can be overkill. Furthermore, it is possible that new algorithms (algorithms not studied in the context of hard real-time systems) are better suited for soft real-time systems (even though their schedulable utilizations are guaranteed to not exceed 50%).

Deficiencies of optimal algorithms. It also follows from the same table that there exist algorithms in the (P_d^u, M_f) class that are optimal, implying that the underlying system need not be underutilized. All known optimal algorithms in this class fall under a subclass called *proportionate-fair* or *Pfair* [25] scheduling algorithms. Nevertheless, the use of optimal Pfair algorithms can be overkill as well for soft real-time systems for the following reasons.

Quantum-based scheduling: Optimal Pfair algorithms are *quantum based*, *i.e.*, schedule tasks one quantum at a time, and as a result, the scheduler is invoked more often and tasks are prone to frequent preemptions and migrations. Hence, the effective system utilization, after accounting for the overheads as explained in Section 1.3.3, can be much less than 100%.

Integer execution costs and periods: Optimal Pfair algorithms require the execution costs and periods of tasks to be specified as integer multiples of the quantum size. Hence, when non-integral, execution costs have to be rounded up and periods have to be rounded down. For some task systems, the associated loss to effective system utilization can be quite significant.

[17] We will omit specifying the task system as a parameter when unambiguous.

Apart from the above restrictions, which can lead to a significant loss of effective system utilization, optimal Pfair algorithms impose some other restrictions also, which can limit their practical implementation and can be problematic for soft real-time systems. Hence, it is desirable that such restrictions, which are described below, be relaxed (apart from relaxing the two restrictions above).

Synchronized scheduling: To ensure optimality, optimal Pfair algorithms require that tasks be allocated processor time in fixed-sized quanta that align across all processors. Apart from posing implementation challenges, this requirement can also lead to wasted processor time in that when a task executes for less than its WCET and yields in the middle of a quantum, or blocks on a lock request in the middle of a quantum, the associated processor idles for the remainder of the quantum. This waste of processor time could potentially be eliminated for soft real-time systems if bounded tardiness can be guaranteed by a simpler algorithm in which the aligned-quanta restriction is relaxed.

Tie-breaking rules: Optimal algorithms require non-trivial tie-breaking rules to resolve ties among subtasks[18] with the same deadline. Such rules may be unnecessary or unacceptable for soft and dynamic real-time systems. Hence, identifying the timeliness properties of algorithms that use *no* tie-breaking rules can enhance the applicability of Pfair algorithms for such systems.

Based on the limitations of optimal and non-optimal algorithms described above, we conclude that the known scheduling algorithms and/or validation tests in all the nine categories are rigid and can be overkill for soft real-time systems. Furthermore, we believe that there is sufficient scope to improve system utilization and implementation support for soft real-time systems on multiprocessors by **(i)** developing new validation tests specifically for soft real-time systems, **(ii)** relaxing some of the limiting restrictions of the optimal Pfair algorithms and deriving the timing properties of the resulting simpler algorithms, and **(iii)** designing new algorithms tailored for soft real-time systems. Accordingly, the thesis of this dissertation is as follows.

> **Thesis Statement:** *Processor utilization can be improved on multiprocessors while providing non-trivial soft real-time guarantees for different soft real-time applications whose preemption and migration overheads can span different ranges and*

[18]In Pfair terminology, each quantum's worth of execution is referred to as a subtask. Refer to Chapter 3 for a more formal definition.

whose tolerances to tardiness are different by designing new algorithms, simplifying optimal algorithms, and developing new validation tests.

Our contributions in support of the above thesis are described below.

1.7 Contributions

We establish our thesis as follows. We analyze two known algorithms, one new algorithm, and some simpler Pfair algorithms that are non-optimal but less expensive than optimal algorithms to determine their soft real-time guarantees. Specifically, for each algorithm $\mathcal{A}$ that we analyze, we determine the guarantees on timeliness that $\mathcal{A}$ can provide if the total system utilization, U_{sum}, exceeds the schedulable utilization, $\mathcal{U}_{\mathcal{A}}(M)$, of $\mathcal{A}$, but not the total available capacity, M, of the underlying system. In the known algorithms category, we show that both global preemptive EDF (*i.e.*, g-EDF) and global non-preemptive EDF (*i.e.*, g-NP-EDF) can guarantee very reasonable, bounded tardiness in most cases. With respect to Pfair algorithms, we show that simpler algorithms that relax one or more of the limiting restrictions described in Section 1.6 that are needed for optimality can guarantee small, bounded tardiness. Finally, we design and analyze a new partitioning-based, restricted-migration algorithm called EDF-fm, which limits the number of tasks that need to migrate, and derive tardiness bounds under it.

The algorithms considered differ in the degree of preemptions and migrations they require, and hence, can benefit different applications that differ in their preemption and migration costs. That is, system utilization levels can be improved for different applications whose overheads span different ranges if they can tolerate bounded tardiness. Similarly, since the algorithms considered differ in the tardiness bounds they can guarantee, different applications with different tardiness thresholds can benefit, as well. Finally, as Block *et al.* show in [36], [35], and [37], the algorithms in the different classes differ in how quickly they can adapt to run-time changes in task composition, *i.e.*, how suitable they are for dynamic task systems. Determining when and how to adapt within any algorithm is quite complex and is beyond the scope of this dissertation. However, in soft real-time systems, rules for adapting to run-time changes require some basic results on timeliness guarantees, which we determine in this research. Thus, the results of this dissertation can enable different applications that differ in the adaptivity they require.

The results we have obtained for soft real-time systems are summarized in Table 1.2 and are discussed in more detail below.

M_f**: full migration**	RM cannot guarantee bounded tardiness for all feasible task systems.	Tardiness is bounded for g-EDF.	Tardiness is bounded for some relaxed Pfair algorithms.
M_r**: restricted migration**	UNKNOWN	Tardiness is bounded for g-NP-EDF. Tardiness is bounded for EDF-fm if $u_{\max} \leq 0.5$.	Tardiness is bounded.
M_p**: partitioned**	Bounded tardiness cannot be guaranteed for task systems that cannot be feasibly partitioned among available processors.		
	P_s**: static**	P_d^r**: restricted dynamic**	P_d^u**: unrestricted dynamic**

Table 1.2: Some tardiness results for the different classes of scheduling algorithms.

1.7.1 Analysis of Preemptive and Non-Preemptive Global EDF

A major contribution of this dissertation is to show that, unlike partitioned-EDF, algorithms g-EDF and g-NP-EDF can guarantee bounded tardiness for sporadic task systems on multiprocessors, and to derive tardiness bounds under them.

If tasks do not block (to access a shared data structure in a mutually-exclusive manner) or suspend (waiting for an I/O operation to complete, for example), as is assumed in this dissertation, then any job can preempt at most one other job under g-EDF. (If blockings and suspensions are not ruled out, then the number of preemptions due to a given job is higher by at most the total number of times the job can be blocked or suspended.) Thus, the total number of job preemptions under g-EDF is bounded from above by the total number of jobs, even though a given job can be preempted more than once. Similarly, in the absence of blockings and suspensions, every job can cause at most one other job to migrate, and the total number of migrations is at most the total number of jobs. The reasoning for this bound on the number of migrations is as follows. Assuming a scheduler that does not migrate a job between two processors if there is a way to execute the job continuously on a single processor while respecting the priority rules, then a preemption by a higher priority job is necessary for a migration. Hence, the bound on migrations follows from the bound on preemptions. For example, in the g-EDF schedule in Figure 1.6(b), the first job of τ_2, $\tau_{2,1}$, migrates twice, first at time 5 and then at time 7. However, these migrations are due to different jobs: the first is due to $\tau_{3,2}$ and the second, due to $\tau_{1,3}$. Thus, because of a moderate number of migrations, g-EDF could be a good choice for applications with moderate migration costs. As discussed in Section 1.4.2.3, partitioning may not be acceptable for highly dynamic task systems even if migration costs are high. The tardiness bound that we have derived under g-EDF on M processors for task T_k of a task system τ with $U_{sum} \leq M$ is given by

$$\frac{\sum_{i=1}^{\Lambda} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda-1} \mu_i} + e_k.$$

(Refer to Section 4.2 for formal definitions of ϵ_i, μ_i, and Λ. Λ is approximately $U_{sum}-1$ and ϵ_i (resp., μ_i) is the i^{th} highest execution cost (resp., utilization) of any task in τ.) As discussed in detail in Chapter 4, the above bound is reasonable unless task utilizations and the number of processors M are both high.

The tardiness bound derived under g-NP-EDF is given by

$$\frac{\sum_{i=1}^{\Lambda+1} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda} \mu_i} + e_k,$$

which is higher than that derived under g-EDF given above. Recall that g-NP-EDF is a restricted-migration algorithm, hence despite being subject to a higher tardiness bound, the algorithm may be preferable when the cost of migrating any given job is high, but not much state is associated with a task itself that is carried between its jobs. Also, on each processor, at most one job may be in a started but unfinished state, and hence, g-NP-EDF has the added benefit of lowering the total stack size, which may especially be desirable in embedded systems. Finally, implementing g-NP-EDF on an SMP is likely to be much simpler than implementing g-EDF, and our experience in implementing these algorithms within the Linux kernel as part of the LITMUS$^{\text{RT}}$ [41] project confirms this. Refer to Chapter 9 for more details.

1.7.2 Design and Analysis of EDF-fm

As a second contribution, we design an algorithm based on partitioned-EDF, called EDF-fm, for use with applications with high inter- and intra-job migration costs. A pure partitioning algorithm (*i.e.*, a no-migration algorithm), offers no scope for guaranteeing bounded tardiness, and hence, for improving processor utilization, for soft real-time systems that cannot be partitioned among the available processors. Thus, *migrations are necessary* in such systems. Hence, in designing EDF-fm, our focus was on minimizing required migrations only to the extent needed for guaranteeing bounded tardiness. Specifically, under EDF-fm, on an M-processor system, the capability to migrate is required for at most $M - 1$ tasks, and it is sufficient that every such task migrates between two processors and at job boundaries only.

1.7.3 Analysis of Non-Optimal, Relaxed Pfair Algorithms

Our third contribution is in determining the timeliness properties of some simpler Pfair algorithms.

Pfair algorithms, which schedule tasks one quantum at a time, may be better suited than EDF for applications with low-migration costs and lower tolerance to tardiness. However, as discussed in Section 1.6, the scheduling and task model restrictions imposed by Pfair scheduling can lower its effective utilization and limit its practical implementation. In an attempt to enhance the applicability of Pfair scheduling for soft real-time systems, we considered relaxing some of those restrictions.

Tie-breaking rules. The *earliest-pseudo-deadline-first* (EPDF) Pfair scheduling algorithm [15] is a non-optimal algorithm, which uses no tie-breaking rules and resolves ties among subtasks arbitrarily. The appropriateness of EPDF for soft real-time systems was first considered by Srinivasan and Anderson in [106], where they derived sufficient per-task utilization restrictions for ensuring bounded tardiness under EPDF. However, they left some questions unanswered. We have answered a few of those questions by extending the existing validation analysis for EPDF in the following ways. First, we show that the prevailing conjecture that EPDF can ensure a tardiness bound of one quantum for all feasible task systems in false. Next, we improve Srinivasan and Anderson's sufficient per-task utilization restrictions for guaranteeing bounded tardiness under EPDF. Third, we derive non-trivial sufficient restrictions along an orthogonal dimension, namely, total system utilization, for schedulability under EPDF, thereby extending the applicability of EPDF for hard real-time systems. Finally, we consider extending the restriction on total system utilization to allow bounded tardiness.

Synchronized scheduling. Elsewhere [50], we have considered relaxing the *synchronized* scheduling requirement of Pfair algorithms, which, as discussed in Section 1.6, requires tasks to be allocated processor time in fixed-sized quanta that align on all processors. We show that if scheduling is *desynchronized*, then under an otherwise-optimal Pfair scheduling algorithm, deadlines are missed by at most the maximum size of one quantum. Further, we argue that this result can be extended to most prior results on Pfair scheduling: in general, tardiness bounds guaranteed under EPDF and other sub-optimal Pfair algorithms are worsened by at most one quantum when scheduling is desynchronized.

Integer execution costs and periods and quantum-based scheduling. We use part of the analysis used in deriving the timing properties of desynchronized Pfair algorithms to show that if periods are non-integral, but execution costs are still integral, then deadlines can be missed by less than two quanta. Finally, to mitigate the loss to total system utilization due to quantum-based scheduling when migration costs are high or execution costs are non-integral, we propose a technique based on job slicing. This job-slicing technique obviates the need to round execution costs up, can lower migration overheads, and can guarantee small and bounded tardiness.

1.7.4 Implementation Considerations and Evaluation of Algorithms

As noted earlier, the performance of each algorithm considered in this dissertation depends on the characteristics of the underlying platform and the application at hand. To better perceive the overheads *versus* schedulability trade-off offered by the various algorithms, we present a simulation-based evaluation of the algorithms using overheads measured on a real test bed and soft real-time schedulability tests. Our evaluation methodology is as follows. We first identify significant sources of external overhead that can delay application tasks, and for each algorithm, provide formulas for computing inflated WCETs for tasks that are sufficient to account for all the overheads. Next, we measure typical values for each overhead on a real test bed, which, in our case, is LITMUS$^{\mathrm{RT}}$, running on a 4-processor SMP [41]. We finally use the measured overhead costs in simulation-based experiments to compute inflated WCETs, schedulability (in a soft real-time sense), and tardiness bounds under the various algorithms considered in this dissertation and under partitioned-EDF for randomly-generated task sets. Our metrics for comparison are the percentage of task systems to which each algorithm can guarantee bounded tardiness and the mean of the maximum tardiness bound guaranteed. For each algorithm, we also discuss scenarios in which it may be the best choice, and briefly comment on efficient implementation techniques.

1.8 Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews prior work on soft real-time scheduling. Chapter 3 reviews Pfair scheduling in detail and presents needed notation. Tardiness bounds under preemptive and non-preemptive global EDF are derived in Chapter 4. In Chapter 5, algorithm EDF-fm and associated analyses for soft real-time systems are presented. A schedulable utilization bound under the EPDF Pfair scheduling algorithm and improved sufficient restrictions on per-task utilizations for bounded tardiness under it are then derived in Chapters 6 and 7, respectively. Supporting non-integral execution costs and periods is considered in Chapter 8, while implementation considerations and a performance evaluation of the concerned algorithms are presented in Chapter 9. Chapter 10 concludes with a summary of the work presented in this dissertation and an enumeration and discussion of some issues that remain to be addressed in the area of soft real-time scheduling on multiprocessors.

Chapter 2

Related Work

In this chapter, prior work on soft real-time systems is surveyed. Work related to the multiprocessor scheduling algorithms considered in this dissertation is discussed in later chapters.

As discussed in Chapter 1, violations of soft real-time constraints are neither disastrous nor desirable, and can be tolerated if they are occasional, or are by bounded amounts. This definition by itself is hardly useful in designing real systems unless the terms "occasional" and "bounded" are precisely specified. Also, if timing constraints can be weakened, then there are different possibilities for doing so, each of which may be more appropriate for some applications than others. Accordingly, several soft real-time models that differ in the nature and extent of violations they allow have been proposed. We will classify such models as either *deterministic* or *probabilistic* based on the predominant nature of the specifications in the model and the type of soft real-time guarantees they provide.

Firm real-time. One type of a non-hard real-time task is a *firm real-time* task. A firm real-time task is one whose deadline misses are not disastrous, but whose results are not useful when late. Thus, it is preferable to not execute a job of a firm real-time task whose deadline will not be met. We will treat a firm real-time task as a special type of a soft real-time task though the two are sometimes considered to be distinct in the literature. Our survey includes models for firm real-time tasks as well.

This chapter is organized as follows. It begins with a survey of deterministic models in Section 2.1. Probabilistic models are considered next in Section 2.2, followed by a discussion of time-value functions in Section 2.3. Time-value functions are not included in Section 2.1 or Section 2.2 because the stated objective of scheduling based on such functions is quite different from those of models considered in these earlier sections. Also, for the general time-value function model, scheduling is predominantly best effort and guarantees are rarely provided.[1] The chapter concludes by enumerating prior work on multiprocessor-based soft real-time scheduling.

2.1 Deterministic Models for Soft Real-Time Systems

In this section, some deterministic soft real-time task models are described. Algorithms for scheduling task systems specified using those models and associated analyses are also described where needed. Unless otherwise specified, all of the work described pertains to uniprocessors.

2.1.1 Skippable Task Model

Among the early deterministic models is the *skippable periodic/sporadic task model* of Koren and Shasha [73]. In this model, a job of a task may be *skipped*, *i.e.*, may not execute at all or have its execution aborted mid-way, as long as there is a minimum separation, referred to as the task's *skip parameter*, between consecutive skips. Task τ_i's skip parameter is denoted s_i, where $s_i \geq 2$, and has the interpretation that for every job of τ_i that is skipped, at least $s_i - 1$ following jobs are not to be skipped. Using Koren and Shasha's terminology, a job that may be skipped is said to be *blue*, and one that has to complete by its deadline, *red*. This model was developed to support some signal-processing systems, where some samples can be skipped, and streaming applications that are tolerant to packet losses.

Let $U_R \stackrel{\text{def}}{=} \sum_{i=1}^{N} \frac{e_i \cdot (s_i - 1)}{p_i \cdot s_i}$ denote the total utilization considering only jobs that may not be skipped. One might reasonably expect that $U_R \leq 1$ is a sufficient feasibility condition. However, consider the task system $\tau = \{\tau_1(3.5, 4), \tau_2(1, 4)\}$. Clearly, this task system is not feasible on a uniprocessor if no job of any task may be skipped. It is also not feasible if only jobs of τ_1 may be skipped regardless of the value of τ_1's skip parameter, s_1. (Note that s_1 should be at least two.) This is because if both τ_1 and τ_2 release a red job each at the same time, then at most one can complete execution by its deadline. For this example, with $s_1 = 2$, *i.e.*, when every other job of τ_1 may be skipped, $U_R = \frac{3.5}{8} + \frac{1}{4} = \frac{5.5}{8}$, which is less than 1.0. Hence, $U_R \leq 1$ is not a sufficient condition for ensuring the feasibility of a skippable task system.

[1] It should be noted that algorithms with provable properties have been developed for some very special cases [28] and providing probabilistic guarantees has been considered in some recent work [79].

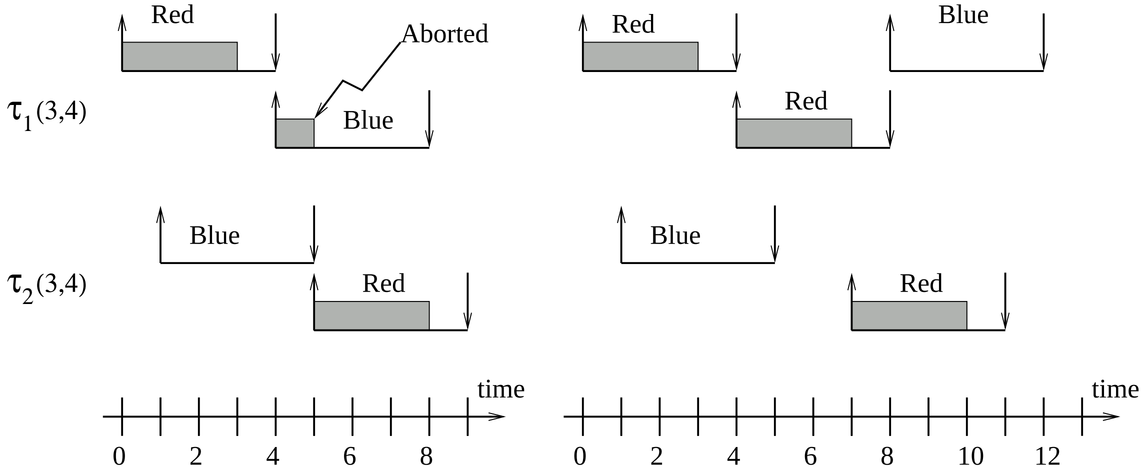


Figure 2.1: Schedules for two concrete instances of a skippable sporadic task system.

As another example, task system $\tau = \{\tau_1(3,4), \tau_2(3,4)\}$ with $s_1 = s_2 = 2$ can be successfully scheduled. If τ is periodic, then a schedule in which odd jobs of τ_1 and even jobs of τ_2, or vice versa, are skipped, would not violate any constraints. Note that for this task system, $U_R = \frac{3}{8} + \frac{3}{8} = \frac{3}{4}$, which is higher that the corresponding value for the previous example. Hence, skippable task systems cannot be compared for feasibility based on their U_R values.

Recall that the model allows jobs to be aborted mid-way and that an aborted job is also considered to be a skipped job. Allowing such behavior may be necessary for sporadic task systems because in such systems, whether a job is red or blue may have to be determined at run-time, based on how jobs of other tasks are released. For illustration, let the task system in the above example be sporadic and consider two partial sets of job releases as shown in insets (a) and (b) of Figure 2.1. In each inset, at most one of $\tau_{1,1}$ and $\tau_{1,2}$ can meet its deadline and $\tau_{1,1}$ is chosen for completion. Therefore, in each inset, $\tau_{2,2}$ is a red instance, and as far as τ_1 is concerned, $\tau_{1,2}$ is skippable, and hence, can be blue. Also, if $\tau_{1,2}$ is skipped, then $\tau_{1,3}$ would be a red instance, whose deadline must be met. $\tau_{1,3}$'s deadline can be met if $\tau_{2,2}$ is released as in inset (a). On the other hand, if the release of $\tau_{2,2}$ is postponed as in inset (b) and $\tau_{1,3}$ is released in time, then at most one of these two jobs can meet its deadline, leading to a timing-constraint violation. However, this scenario can be avoided because $\tau_{2,2}$'s release is sufficiently delayed that $\tau_{1,2}$'s deadline can be met and $\tau_{1,3}$ can be turned into a blue instance. Therefore, when tasks are sporadic, schedulability can be improved by dynamically choosing to not skip certain jobs.

It turns out that optimally determining which jobs to skip is NP-hard even if all job releases are known *a priori*. Hence, Koren and Shasha proposed the use of algorithms based on simple heuristics. In their first algorithm, called *red tasks only* (RTO), every $(k \cdot s_i)^{\text{th}}$ job, for every $k \geq 1$, is considered blue, and is skipped. The remaining jobs are scheduled by either EDF or RM. Note that the task system in the second example above is not schedulable by RTO, even if it is periodic. Their second algorithm is called *blue when possible* (BWP), and tries to provide improved service in comparison to RTO in a best-effort manner by scheduling some blue job (chosen according to one of several heuristics) when no red job is pending. They also provided associated schedulability conditions for both EDF and RM.

2.1.2 (m, k)-Firm Model

The (m, k)-*firm model* is similar to, and, in fact, subsumes the skippable task model, and was proposed by Hamdaoui and Ramanathan [62].[2] In this model, each task is associated with two parameters m and k with the interpretation that at least m jobs in every window of k consecutive jobs must meet their deadlines. Note that a skippable task with a skip parameter s is also an (m, k)-task with parameters $m = s - 1$ and $k = s$, and that while every skippable task can be represented as an (m, k) task the converse does not hold. Also, the (m, k) model can allow skips to be consecutive, and in that sense, can provide weaker guarantees than the skippable model.

Hamdaoui and Ramanathan proposed an algorithm called *distance based priority* (DBP) for scheduling (m, k)-tasks. Unlike Koren and Shasha's algorithms, scheduling under DBP is dependent on run-time conditions. Hence, DBP maintains a state, which captures the current history of met and missed deadlines, with each task. The task states are used to determine how "unsafe" each task is, *i.e.*, how close it is to having its (m, k) constraint violated, and accord higher priority to tasks that are more unsafe. Tasks within the same priority class are scheduled on an EDF basis.

The goal of DBP is to schedule tasks so that the average probability of violating the (m, k) constraints is minimized and the realization of this goal is evaluated in [62] through simulations. The model and the algorithm were initially developed for scheduling streams of network packets, and hence, in the evaluations, the arrival pattern for jobs is neither periodic nor sporadic, but is either Poisson or bursty. In [63], an analytic model of DBP that can be

---

[2] It appears that the skippable and (m, k) models were proposed at about the same time.

used to provide probabilistic guarantees on meeting the (m, k) constraint is developed, and its accuracy is evaluated through simulations.

The application of the (m, k) model to overloaded periodic task systems is considered in [98]. Here, as with RTO, static rules are provided for skipping jobs, and unskipped jobs are scheduled using RM. A sufficient condition for deterministically meeting (m, k) constraints is also derived.

2.1.3 Weakly-Hard Model

The *weakly-hard* model [32] has elements of both of the models described above and strives to provide a general framework for the specification of soft real-time constraints. In [32], soft real-time constraints are referred to as *weakly-hard constraints*. Currently, four types of constraints, denoted $\binom{n}{m}$, $\left\langle {n \atop m} \right\rangle$, $\overline{\binom{n}{m}}$, and $\overline{\left\langle {n \atop m} \right\rangle}$, can be specified. The parameter m in all of these expressions refers to any window of m consecutive jobs; parameter n refers to either the number of met or missed deadlines. The $\binom{n}{m}$ specification is simply the (m, k) specification and means that at least n jobs in any window of m consecutive jobs must meet their deadlines. This is strengthened by $\left\langle {n \atop m} \right\rangle$, which requires n consecutive deadlines to be met. The specifications with overbars apply to the number of missed deadlines: $\overline{\binom{n}{m}}$ means that fewer than n jobs out of any consecutive m jobs may miss their deadlines; $\overline{\left\langle {n \atop m} \right\rangle}$ is a stronger specification under which, out of m jobs, any number may be missed as long as fewer than n are consecutive.

In [32], the authors also developed an algebra of weakly-hard constraints that can allow different types of constraints and constraints with different values for n and m to be related and compared. They also discussed the possibility of associating multiple constraints with each task. For instance, if a task can tolerate at most two misses in any window of four jobs, but only at most eight misses (not ten) in a window of twenty jobs, then such a constraint can be specified as $\binom{2}{4} \wedge \binom{8}{20}$.

Finally, the scheduling of weakly-hard real-time systems under static-priority algorithms is considered. No special rules are included to improve the chances of meeting weakly-hard constraints; rather, the objective is to determine whether a given system can meet all the constraints when scheduled using static priorities. This can be done in a straightforward manner by using standard static-priority analysis for computing the response times[3] of jobs released

[3]The *response time* of a job is given by the difference between the time the job completes executing and its release time.

within a hyperperiod. The authors showed that neither the RM nor *deadline-monotonic*[4] priority assignment is optimal for weakly-hard systems and discussed how to optimally assign priorities.

2.1.4 Window-Constrained Model

The *window-constrained model* is a weaker variant of the (m, k)-firm model and was first considered in [119] and [118]. In this model, windows are non-overlapping in that every group of k jobs from the beginning forms a window. In other words, in this model, jobs $(k(i-1)+1), \ldots, k \cdot i$, comprise the i^{th} window, where $i \geq 1$, whereas under the (m, k) model, jobs $i, \ldots, (i+k-1)$ form the i^{th} window. The model parameters are specified as a pair (x, y) with the meaning that at most x jobs in every non-overlapping window of y jobs may miss their deadlines. In [119] and [118], an online algorithm called *dynamic window-constrained scheduling* was proposed for scheduling packet streams with window constraints in communication networks. Conditions for scheduling streams without violating the window constraints are also derived under the assumption that packets are uniform in size and are non-preemptive.

The window-constrained model was subsequently studied by Mok and Wang [92], who showed that, in general, determining whether a schedule that satisfies the window constraints is possible is NP-hard in the strong sense. They also showed that some sub-classes of the problem (such as those with unit execution costs, or are synchronous, periodic, or for which $x \mid y$ holds) can be solved efficiently, and devised scheduling algorithms and/or schedulability tests for those sub-classes.

2.1.5 Imprecise Computation Model

The models discussed above allow some jobs to be treated as optional and their executions to either complete late or be discarded entirely, as long as the optional jobs conform to the constraint specification. The *imprecise computation model* takes a different approach: under this model, some portion of each job may be discarded; however, in general, no job may be discarded in its entirety or complete late. This model was proposed to allow graceful degradation during overload for tasks based on iterative algorithms such as numerical computations or heuristic search, some multimedia tasks such as layered encoding and decoding, and others.

[4] The deadline monotonic algorithm is a static-priority algorithm, which accords higher priority to tasks with shorter relative deadlines.

For such tasks, the quality of the results improves with the amount of time for which the jobs execute; however, a certain minimum acceptable quality is reached when only a fraction of the job completes execution. Hence, under this model each job of a task τ_i is logically decomposed into two parts: a *mandatory* part, with execution cost e_{m_i}, that corresponds to the minimum required computation for the result to be acceptable, and an *optional* part, with execution cost e_{o_i}, whose execution improves the quality of the results. The result produced by a job is *precise* only if its mandatory and optional parts are both executed; otherwise, it is *imprecise*. The optional part of each task is also associated with a *reward function* (or, correspondingly, an *error function*), which is non-decreasing (resp., non-increasing) with respect to execution time, and for a given time t, indicates the value (resp., penalty) that the system accrues by executing (resp., not executing) the optional part of a job for t (resp., $e_{o_i} - t$) time units before its deadline. The goal is to schedule tasks such that the mandatory part of each job completes by its deadline and the optional parts are executed such that some performance metric, such as the number of optional parts that are discarded or the average or total error, is minimized.

The imprecise computation model was considered by Liu *et al.* in [83], where they proposed heuristic algorithms for scheduling such task systems. To ensure that the mandatory parts never miss deadlines, the algorithms they proposed accord higher priority to mandatory jobs and schedule them under either EDF or RM. Optional parts are executed when no mandatory part is pending. The different heuristics differ in how optional jobs to execute are chosen. Liu *et al.* allow optional parts to complete late.

This problem was later addressed by Aydin *et al.* [19], who considered optimal schedules in which the average reward is maximized. They showed that if the reward functions are linear or concave, then for each periodic task system whose mandatory parts can be successfully scheduled, there exists an optimal EDF or LLF schedule in which, for each task, the optional part of each of its jobs executes for equal times. They also showed that the execution times for optional parts can be determined by solving an optimization problem and presented efficient methods for doing so. Finally, they also showed that the problem of determining an optimal schedule is NP-hard if the reward functions of tasks are non-concave.

2.1.6 Server-Based Scheduling

For some tasks, minimum inter-arrival times and/or worst-case execution times that are not too pessimistic cannot be determined, and hence, such tasks do not fit any recurrent task model such as the periodic or the sporadic model. Jobs that execute in response to mode change requests and effect mode changes, such as a job that changes the operation mode of autopilot, or jobs in a command and control system that respond to sporadic data messages are some examples [85]. One common and popular approach for scheduling such non-recurrent tasks, also referred to as *aperiodic tasks*, is by using *server tasks*. A server task is a periodic or sporadic task that is scheduled along with other recurrent real-time tasks in the system and is used to serve aperiodic tasks; whenever a server task is scheduled, its allocations are passed on to one of the pending aperiodic tasks assigned to it. Each server task is assigned an execution cost, referred to as its *budget*, and a period, both of which are determined based on the needs of the aperiodic tasks its serves. The schedulability of the recurrent tasks and server tasks is guaranteed offline. The aperiodic tasks may have hard deadlines, in which case, they are subject to admission-control tests upon release. Since our interest is in scheduling soft real-time tasks, we will not consider scheduling hard aperiodic tasks.

In general, in the server-based approach, soft real-time tasks are not associated with any timing constraints, and the objective is to improve their response times in a best-effort manner. If the server queues are always backlogged and hard real-time tasks execute for their full worst-case execution times, then there cannot be much scope to improve response times of soft aperiodic tasks. However, the above conditions rarely hold in practice, and hence, there arises an opportunity to intelligently reclaim any capacity that is made available to improve soft real-time response times. The challenge, however, is to ensure that the safety of hard tasks is not compromised. Techniques for doing so have been extensively investigated and a huge body of literature devoted to the topic is available [78, 44, 101, 60, 45, 103, 80, 40, 88, 31].

2.1.7 Maximum Tardiness

The soft real-time model considered in this dissertation also provides deterministic guarantees. This model, where sporadic tasks with implicit deadlines have a tardiness threshold, has not been considered much in the context of uniprocessors. This is because, if the system is not overloaded, then it is obvious that scheduling under EDF is sufficient to ensure that each job completes execution by its deadline, and that there is no scope of allowing a higher utilization even if bounded tardiness is tolerable. This model can alternatively be viewed as one in which timing constraints are hard, but sporadic tasks have relative deadlines *larger* than their periods.[5] When viewed in this alternative manner, the model has been the focus of some research in the context of deadline-monotonic scheduling on uniprocessors. In [75], Lehoczky provided a demand-based test, which may require exponential time, to determine the schedulability of task systems that use this model, and also derived utilization bounds when the relative deadline of each task is a multiple (greater than one) of its period. However, determining the maximum time after the end of its period that any job may complete executing has not been addressed. Such a value can be determined in a straightforward manner using the generalized time-demand analysis developed for static-priority systems.

We believe that the models and work discussed above provide a reasonable sampling of deterministic soft real-time guarantees that have been considered. We now move on to discuss some stochastic models that have been proposed.

2.2 Probabilistic Models for Soft Real-Time Systems

Some tasks with widely varying inter-arrival times or execution requirements, such as those in video-conferencing applications or animation games, require soft real-time guarantees such as bounds on tardiness or limits on the percentage of deadlines missed. Hence, it may not be appropriate to schedule such systems using the server-based approaches described earlier in which a single server serves more than one task or in which needed guarantees cannot be provided. For such systems, one alternative to reserving resources based on worst-case parameters, which can be extremely wasteful, is to model task parameters probabilistically (using probability distributions for inter-arrival times or execution times) and to provide probabilistic guarantees on meeting deadlines. In this section, we describe some efforts taken in this direction.

2.2.1 Semi-Periodic Task Model

In [115], Tia *et al.* considered scheduling tasks whose jobs have highly varying execution requirements but are released periodically. Tia *et al.* referred to such tasks as *semi-periodic tasks* and characterized their execution requirements using generic probability density functions (pdfs). They then extended the time-demand and generalized time-demand analyses [77, 75]

[5] It should be noted that if the relative deadlines of all tasks are not increased by the same amount, then job priorities may be determined differently when viewed in this alternative manner.

used with static-priority algorithms for semi-periodic task systems under the assumption that the worst-case response time for each task can be found in a busy interval that begins when every task releases a job. Whereas standard time-demand analysis simply sums the demand due to individual jobs that execute in a busy interval, in the probabilistic analysis proposed by them, a convolution of the pdf's of the execution times of the jobs is taken. The outcome of the convolution is a pdf for the total demand at any time within the busy interval. This pdf can then be used to compute the probability that a given job's response time exceeds its deadline. Tia *et al.* noted that their approach is valid only if job execution times are independent and discussed how to correctly compute failure probabilities in the absence of this independence assumption.

In the same paper, an alternative approach called *task transformation* is proposed for scheduling semi-periodic tasks. The intention is to guarantee 100% schedulability for short jobs and some fraction of each long job. Hence, it is proposed that each job of a semi-periodic task be logically decomposed into two components: a fixed-size periodic component and a variable-size sporadic component. The execution requirement is constant for the periodic components of all the jobs and may vary for the sporadic components. Also, the periodic components are guaranteed deterministically, whereas the sporadic components are served by special server tasks of appropriate utilizations and are provided probabilistic guarantees. (A single server task may serve multiple sporadic component tasks.) Note that, in some sense, the periodic and sporadic components are reminiscent of the mandatory and optional parts of the imprecise computation model.

The semi-periodic task model was also considered by Diaz *et al.* in [55]. However, the analysis they provided is different from that of Tia *et al.*. Diaz *et al.* modeled the varying execution costs using discrete random variables characterized by probability mass functions. They showed how the state of a priority-driven system can be modeled as a discrete-time Markov chain (DTMC) and how a response-time distribution may be determined (both numerically and analytically) using it.

2.2.2 Statistical Rate-Monotonic Scheduling

The goal of *statistical rate-monotonic scheduling* (SRMS) proposed by Atlas and Bestavros in [18] is to schedule periodic task systems such that the statistical quality-of-service (QoS) guarantee needed by each task is met. The task model considered in this work is similar to and is based on the semi-periodic model described above, and associates with each task τ_i

a constant inter-arrival time, a pdf characterizing the utilization of its jobs (using which the execution requirement may be computed), and a permissible QoS. The QoS of a task is defined as a lower bound on the probability that an arbitrary job will meet its deadline.

As explained in [18], the main tenet of SRMS is that variability in task execution requirements can be smoothened by aggregating the requirements of successive jobs. SRMS is based on RM, and for each task τ_i, treats jobs that fit within a *superperiod* as a unit. The length of τ_i's superperiod is given by that of the period of the next lower-priority task, τ_{i+1}. Under RM, at most $\lceil \frac{p_{i+1}}{p_i} \rceil$ jobs of τ_i can interfere with any job of τ_{i+1}. Also, τ_{i+1} is not impacted by *how* the aggregate cost is distributed among τ_i's jobs as long as the distribution is among the jobs within a superperiod. SRMS makes use of these properties to ensure that statistical guarantees are met. Hence, at run-time, for each task, allocations made to jobs in its current superperiod are maintained; this information, along with the resource needs of all higher-priority tasks, is used to determine whether the deadline of a newly arriving job can be met. (It is assumed that the execution requirement of a job is known when it is released.) A job whose deadline cannot be met is simply rejected, and thus, every job that is admitted is guaranteed to meet its deadline. Offline probabilistic analysis ensures that the percentage of discarded jobs does not violate the QoS requirement of the task.

The main differences between SRMS and the approaches considered in Section 2.2.1 are that in SRMS, only jobs guaranteed to meet their deadlines are admitted to the system, and higher-priority tasks cannot overrun and infringe on lower-priority tasks. One critique of this model, however, is that the assumption that job execution requirements are known at their release times is somewhat questionable.

2.2.3 Constant-Bandwidth Server

In the work considered so far in this section, only execution costs of the jobs of a task are allowed to vary. Tasks whose job inter-arrival times may also vary were considered by Abeni and Buttazzo in [5], [6], and [7]. Unlike the previous approaches, Abeni and Buttazzo considered scheduling each variable-parameter task using a separate, dedicated server, referred to as a *constant bandwidth server* (CBS). Recall that a server is a periodic or a sporadic task with a budget (*i.e.*, execution cost) and a period. Under CBS, a server task's budget and period are set to the mean execution cost and the mean inter-arrival time, respectively, of the task it serves. Server rules are defined such that, in the long run, each client task is allocated a fraction of the total processor time approximately equal to its mean utilization, and the different clients are temporally isolated[6] from one another. Thus, in effect, an appropriate fraction of the processor is *reserved* for each variable-parameter task.

Abeni and Buttazzo also presented analysis for tasks scheduled using CBS servers. Their analysis can, in general, be applied to any reservation-based system. However, their analysis is restricted to allowing only one of the two parameters (either execution cost or inter-arrival time) to vary. They showed how to model the state of a CBS-served task with one varying parameter as a discrete-time Markov chain, and how to compute a steady-state probability for any given tardiness for the task. Because each task is reserved a fraction of a processor, they were able to model each task as an independent stochastic process and analyze it independently, without consideration of interference from the other tasks. In our opinion, this considerably simplifies their analysis in comparison to those discussed for semi-periodic tasks in Section 2.2.1.

2.2.4 Real-Time Queueing Theory

The probabilistic analyses that we have considered so far allow at most one parameter of each real-time task to be stochastic. In contrast, the grand aim of *real-time queueing theory* seems to be to allow every aspect of the traditional real-time system model to be stochastic, if needed, and develop tools for analyzing such systems scheduled under priority-driven algorithms. Real-time queueing theory strives to achieve this aim by combining the timing elements of real-time scheduling theory with the stochastic elements of queueing theory. This theory was first proposed by Lehoczky in [76], where a semi-formal analysis is presented for systems with a single queue under heavy-traffic conditions.[7] (Each queue corresponds to a task, so a single-queue system is essentially a system with a single task serving aperiodic jobs.) Fully-developed theory for single-server, single-queue systems scheduled under EDF and FIFO is presented in [56], and that for acyclic networks of servers with multiple independent queues in [74]. It should be noted that apart from inter-arrival times and service times, job deadlines are also modeled as independent and identically distributed random variables. However, it is not clear whether the assumption that all times are independent is realistic, and what removing this assumption entails.

[6] A set of tasks is said to be *temporally isolated* if execution overruns of any task cannot impact any of the remaining tasks.

[7] Under heavy traffic conditions, the traffic intensity or the average processor utilization converges to one. According to Lehoczky, the heavy-traffic case is the worst case for real-time systems, and hence, can be used as an upper bound for lighter conditions.

One critique of the probabilistic models by proponents of other models is that percentage of deadline misses or probability of meeting a deadline by itself is not sufficient to assess quality of service unless the distribution of misses is also known. With that note, we conclude our discussion on probabilistic models.

2.3 Time-Value Functions

Many soft real-time models are implicitly based on the notion that in some systems, a late result is still of value, although diminished, in comparison to a timely result. However, the extent to which the value is diminished is not quantified. Jensen, Locke, and Tokuda's *value functions* make this notion more precise by associating with each task, an explicit value, which is expressed as a function of completion time [72]. A task's value at time t indicates the value that the task adds to its system if it completes executing t time units after its release. Jensen *et al.* also suggested determining task deadlines or *critical times* from value functions. They contended that a task can be said to have a deadline only if its value function or its first or second derivative is discontinuous and that the deadline is given by the earliest time at which a discontinuity occurs. Hence, the value function for a traditional hard real-time task can be given by a downward step function with the step at its deadline as shown in Figure 2.2(a). The figure also shows some other examples of value functions as provided in [72].

Jensen *et al.* also contended that the goal of real-time scheduling should be to execute tasks such that the total value accrued by the system is maximized. They proposed algorithms based on heuristics for this purpose and evaluated them in comparison to traditional algorithms under different levels of load, including overloads. In general, under this model, accrued value can be maximized by scheduling under EDF if all deadlines can be met. However, when the system is overloaded and not all deadlines can be met, minimizing deadline misses will not necessarily increase accrued value, and heuristics perform much better. There has been some renewed interest in value-based scheduling in recent years and the interested reader is referred to [79]. One major impediment to the use of this model is the difficulty associated with defining value functions themselves. Case studies of successful applications of this approach can be found in [71].

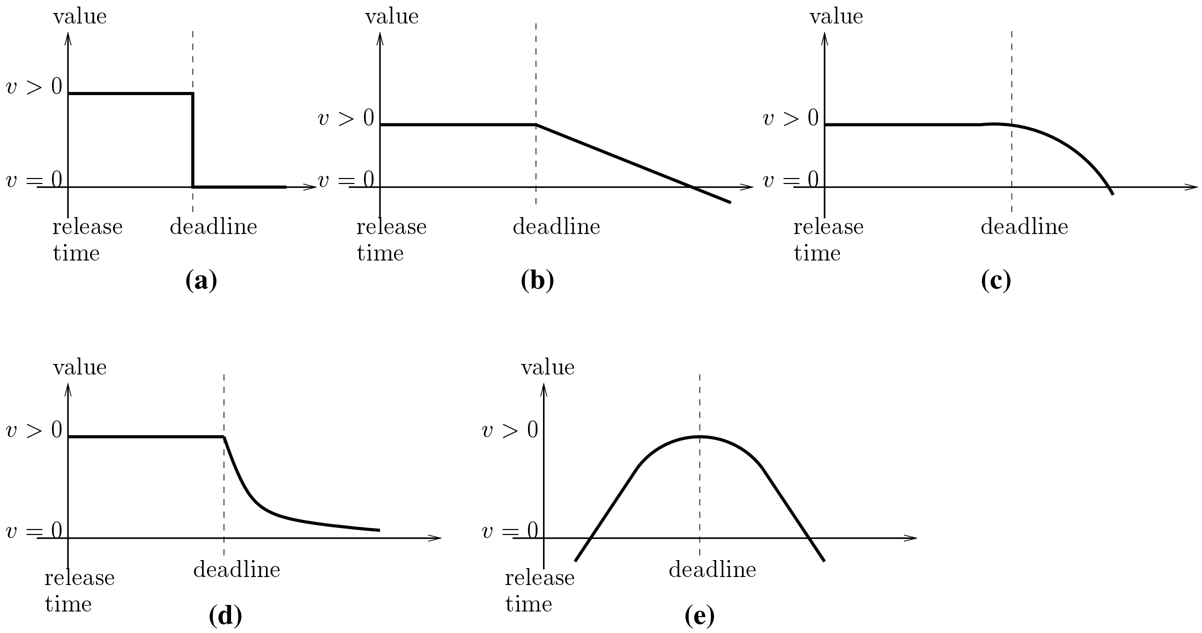


Figure 2.2: Examples of value functions.

2.4 Soft Real-Time Scheduling on Multiprocessors

To our knowledge, scheduling recurrent, soft real-time systems on multiprocessors was first considered by Srinivasan and Anderson in [106], where they studied the appropriateness of the non-optimal but more efficient EPDF Pfair scheduling algorithm for such systems. Other work that falls under this category consists of the server-based approaches for scheduling aperiodic tasks considered in [110], [27], [30], and [29]. The evaluation of Jensen *et al.* mentioned in Section 2.3 includes multiprocessors, but they allow different jobs of a task to execute concurrently, and hence, the evaluation cannot be considered to be truly for recurrent tasks on multiprocessors. Finally, though the real-time queueing theory presented in [74] is for networks of single-server stations, it cannot support even a single station with multiple servers.

2.5 Summary

In this chapter, we surveyed some prior work on soft real-time systems. We classified soft real-time models as either deterministic or probabilistic, based on the predominant nature of the specifications in the model and the type of soft real-time guarantees provided, and under each class, surveyed a representative sampling of prior work. We then briefly discussed scheduling based on time-value functions, and finally considered prior work on soft real-time scheduling on multiprocessors. From the survey presented in this chapter, it is quite evident that soft real-time scheduling on multiprocessors is largely an unexplored area of research. In the following chapters, we explore one part of it.

Chapter 3

Background on Pfair Scheduling

In Chapters 6, 7, and 8, we consider scheduling soft real-time systems using Pfair algorithms. Before embarking on that endeavor, this chapter describes some basic concepts of Pfair scheduling, provides needed background, and summarizes relevant results from prior work. Since most prior work on Pfair scheduling has been for hard real-time systems, most of the discussion in this chapter is with respect to such systems.

The chapter is organized as follows. Section 3.1 motivates the concept of Pfair scheduling. The basic elements of Pfair scheduling are described in the context of synchronous, periodic task systems in Section 3.2. Section 3.3 then describes other task models that can be Pfair-scheduled. Pfair scheduling algorithms are discussed afterwards in Section 3.4, followed by brief descriptions of some extensions that have been proposed to enhance the practicality of the Pfair approach. The chapter concludes after summarizing some definitions and results from prior work that are somewhat technical in nature and that are used in the analyses presented in later chapters.

3.1 Introduction

In most of the algorithms described so far, jobs are first-class objects in the sense that they are the basic units of work. Furthermore, timing constraints (*e.g.*, deadlines or tardiness requirements) are associated with jobs. How job fragments execute is, in general, of little concern.

This job-oriented paradigm proved sufficient for the optimal online scheduling of periodic

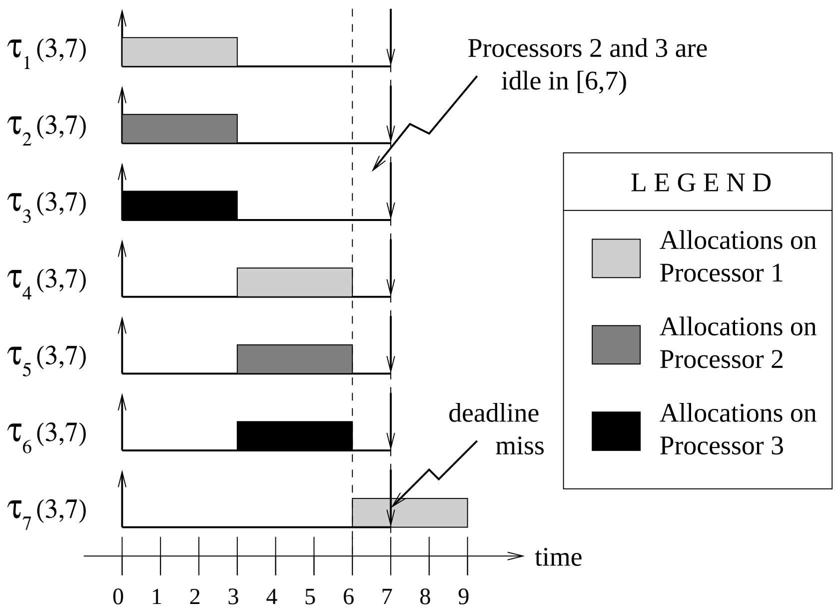


Figure 3.1: A g-EDF schedule with a deadline miss for a task system that is feasible on three processors.

and sporadic real-time task systems on uniprocessors in polynomial time;[1] however, it has so far proved to be inadequate for constructing such schedules on multiprocessors. In what follows, we will try to illustrate why this is so.

We will begin by considering the example in Figure 3.1. In this example, seven tasks, each with an execution cost of three time units and a period of seven time units, are scheduled under g-EDF on three processors.[2] A schedule for the first job of each task is shown. Here, the jobs of $\tau_1, \ldots, \tau_6$ complete executing by time six, whereas that of τ_7 does not execute at all until time six and its deadline is at time seven. Though the cumulative processing capacity available in $[6, 7)$ is sufficient to meet the needs of τ_7's job, restrictions on concurrency prevent it from using all but one processor, and it misses its deadline at time 7 with two units of computation pending. It is appropriate to note that, as remarked by Liu in [81], and echoed by Baruah *et al.* in [25], "the simple fact that a task can use only one resource even when several of them are free at the same time adds a surprising amount of difficulty to the scheduling of multiple resources."

[1]The time complexity of an online scheduling algorithm is given by the time required to make a single scheduling decision.

[2]Note that the task system in this example cannot be partitioned among three processors, and hence, cannot be scheduled correctly under any partitioning-based algorithm.

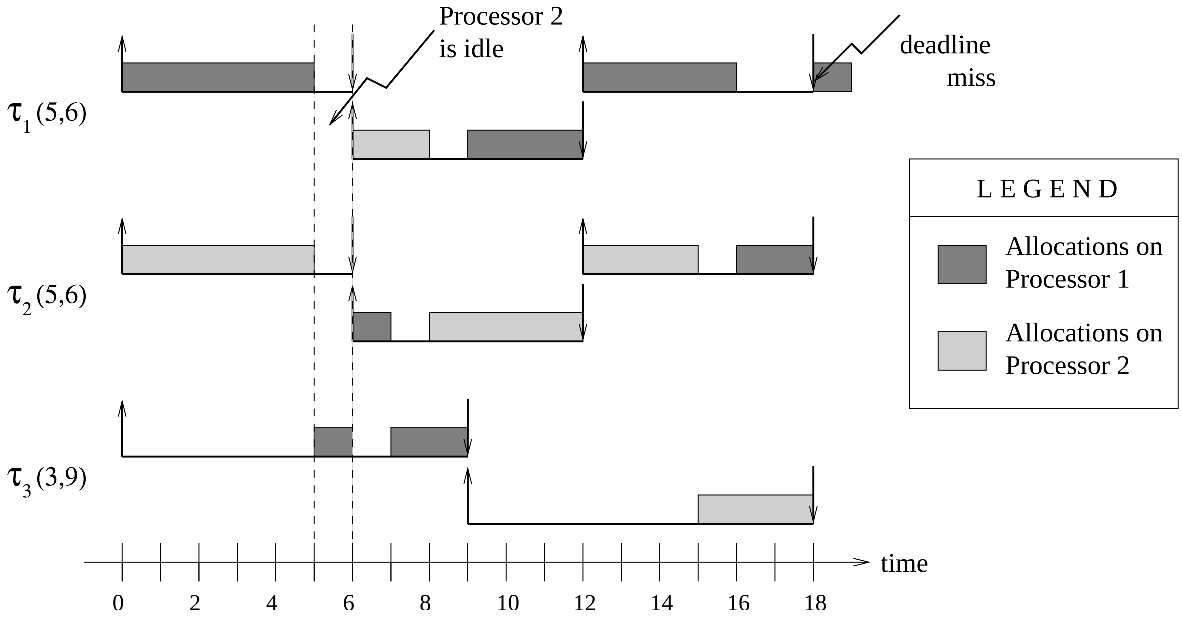


Figure 3.2: An LLF schedule with a deadline miss for a task system that is feasible on two processors.

It is clear that τ_7's deadline miss can be averted by appropriately substituting its execution for those of two of the other tasks, and one may be tempted to attempt using the LLF scheduling algorithm to fix the problem. Though LLF can avert the deadline miss in the example above, it is not optimal either. An example of a feasible task system that misses a deadline under LLF is provided in Figure 3.2. In this example also, there exists a time interval (which is $[5, 6)$) where a processor is idled due to concurrency restrictions, which in turn leads to the deadline miss indicated in the figure. Note that idling in this case can be avoided by postponing the execution of either $\tau_{1,1}$ or $\tau_{2,1}$ by one time unit in exchange for executing $\tau_{3,1}$ earlier. However, such ad-hoc rules cannot solve this noted problem for any arbitrary task system, and until the work of Baruah *et al.*, which is described below, it was doubtful whether systematic rules could be devised at all. In fact, fueled by some negative results, including one by Dertouzos and Mok in [47], which states that there does not exist an optimal algorithm for scheduling a set of one-shot, real-time tasks[3] on a multiprocessor if not *all* the release times, execution costs, and deadlines of all tasks are known *a priori*, the problem of optimally scheduling periodic and sporadic tasks online on a multiprocessor was believed to be NP-hard, and the focus seemed to be on developing efficient heuristics [95, 113].

[3]A one-shot, real-time task releases a single job. The sole job is associated with a release time, execution requirement, and a deadline.

Finally, in [25], in their seminal work on *Pfair scheduling*, Baruah, Cohen, Plaxton, and Varvel showed that the problem can be elegantly solved in polynomial time by strengthening the timeliness constraints of tasks (provided some restrictions, which can alternatively be viewed as rules of their scheduling approach and which are described in the next section, are adhered to). Their solution is explained in detail in the next section. In essence, in their solution, job deadlines are met by requiring tasks to execute at approximately uniform rates given by their utilizations at all times. This is achieved by subjecting each time unit's worth of execution of each task, later termed a *subtask* by Anderson and Srinivasan [15], to a deadline, and ensuring that subtask deadlines are met. Thus, in Pfair scheduling, the notion of job is subordinate, and subtasks are first-class objects.

Notational differences. Before considering Pfair scheduling in detail, we would like to alert the reader of the differences in notation with respect to that used in chapters not concerned with Pfair scheduling. In the context of Pfair scheduling, tasks are denoted using upper case letters near the end of the alphabet, such as T, U, and V. (This was mentioned earlier in Section 1.3.1.) The period and execution cost of a task T are denoted $T.p$ and $T.e$, respectively. The utilization of T is referred to as its *weight* and is denoted $wt(T) \stackrel{\text{def}}{=} T.e/T.p$.

3.2 Synchronous, Periodic Task Systems

In this section, some basic Pfair concepts are explained in the context of synchronous, periodic task systems. We will simply refer to such task systems as periodic, omitting the term synchronous.

As mentioned earlier, under Pfair scheduling, tasks are required to execute at rates that are more uniform over time than that mandated by periodic scheduling. For instance, in the periodic schedule for τ_1 in Figure 3.1, though the execution rate of τ_1 is $3/7$ when averaged over the interval $[0, 7)$, there are significant variations over shorter intervals; in particular, execution rates during subintervals $[0, 3)$ and $[3, 7)$ are 1.0 and 0.0, respectively. The ideal case would be to execute every task T of τ at the rate of $wt(T)$ over any infinitesimal interval. Such ideal execution for a task T with $wt(T) = 3/7$ is shown in Figure 3.3(a). Such an ideal schedule is also referred to as a *fluid schedule* or a *processor-sharing schedule*. In practice, such ideal schedules can be realized only if task contexts can be switched at arbitrarily fine time scales, and hence, are not achievable. Assuming that restrictions (or rules) (R1) and (R2) below hold, Baruah *et al.* showed that ideal execution is not necessary for solving the problem at hand, and an approximation that closely tracks the ideal is sufficient.

(R1) Tasks are allocated processor time in discrete time units or *quanta* that are uniform in size and align on all processors.

(R2) The execution cost and period of each task are integer multiples of the quantum size.

Specifically, Baruah *et al.* showed that when (R1) and (R2) hold, *all* the job deadlines of a feasible periodic task system can be met even if at any time t, the absolute deviation in the total allocation that each task receives up to t, with respect to the ideal schedule, that is, its absolute lag, is arbitrarily close to one time unit. They formally defined a schedule in which (3.1) below holds to be a *proportionate-fair schedule*, or, in short, a *Pfair schedule*, for τ, showed that in such a schedule, all of τ's job deadlines are met, and presented an online algorithm for constructing Pfair schedules.

$$(\forall t, T \in \tau :: -1 < \mathsf{lag}(T, t, \mathcal{S}) < 1) \tag{3.1}$$

The formula for lag is given below in (3.3) after notational conventions and the implications of the above assumptions are set in place.

Unless otherwise specified, (R1) and (R2) above are assumed to hold. System start time is zero, and for all $t \in \mathbf{N}$, where $\mathbf{N}$ is the set of natural numbers,[4] the real time interval spanning $[t, t+1)$, *i.e.*, the real time between time t and time $t+1$ that includes t and excludes $t+1$, is referred to as *slot* t. Unless otherwise specified, in the context of Pfair scheduling, all references to time are integral. The quantum size equals the slot size, and on each processor, the $i+1^{\text{st}}$ quantum is allocated in slot i.[5] The interval $[t_1, t_2)$, consists of slots $t_1, t_1+1, \ldots, t_2-1$. To ensure that (R1) holds, scheduling decisions are made only at slot boundaries. In each slot, each task can be scheduled on at most one processor, and on each processor at most one task can execute. However, a task can be assigned to different processors in different slots. That is, a task may migrate across processors, but may not execute concurrently.

An ideal or processor-sharing schedule for τ shall be denoted PS_τ; the subscript τ will be omitted when the task system under consideration is unambiguous. An actual schedule $\mathcal{S}$

[4] We assume that $\mathbf{N}$ includes zero.

[5] Quantum and slot are often used interchangeably.

for a task system τ is a sequence of allocation decisions over time and is given by a function $\mathcal{S} : \tau \times \mathbf{N} \to \{0, 1\}$. If task T is scheduled in slot t, then $\mathcal{S}(T, t) = 1$; otherwise, $\mathcal{S}(T, t) = 0$. Because at most one task can be scheduled in each slot on any processor, $\sum_{T \in \tau} \mathcal{S}(T, t) \leq M$ holds. Also, each task that is scheduled in slot t is allocated one time unit in that slot. Hence, for all integral t_1 and t_2, where $t_2 > t_1$, $\mathsf{A}(\mathcal{S}, T, t_1, t_2) = \sum_{u=t_1}^{t_2-1} \mathcal{S}(T, u)$ holds, where $\mathsf{A}(\mathcal{S}, T, t_1, t_2)$ denotes the total allocation to task T in $\mathcal{S}$ in the interval $[t_1, t_2)$. As a short hand, we will let $\mathsf{A}(T, \mathcal{S}, t)$ denote the allocation to T in $\mathcal{S}$ in slot t. Therefore, for integer t, $\mathsf{lag}(T, t, \mathcal{S})$, which denotes the difference between the total allocations that T receives in the interval $[0, t)$ in PS and $\mathcal{S}$, is given by

$$
\begin{aligned}
\mathsf{lag}(T, t, \mathcal{S}) &= \mathsf{A}(\mathsf{PS}, T, 0, t) - \mathsf{A}(\mathcal{S}, T, 0, t) \\
&= \sum_{u=0}^{t-1} \mathsf{A}(\mathsf{PS}, T, u) - \sum_{u=0}^{t-1} \mathcal{S}(T, u) \qquad (3.2) \\
&= wt(T) \cdot t - \sum_{u=0}^{t-1} \mathcal{S}(T, u) \qquad (3.3) \\
&\quad \text{(because the ideal allocation to a periodic task is } wt(T) \text{ in every slot).}
\end{aligned}
$$

Subtasks, pseudo-releases, and pseudo-deadlines. As already mentioned, in Pfair terminology, each quantum of execution of each task is referred to as a *subtask*. Hence, each task consists of an infinite sequence of subtasks. The i^{th} subtask of T is denoted T_i, where $i \geq 1$. Therefore, letting $e = T.e$, by (R2), the k^{th} job of T consists of subtasks $T_{(k-1)\cdot e+1}, \ldots, T_{k \cdot e}$.

We now explain how to determine the interval within which a subtask needs to execute in a Pfair schedule. In following the ensuing discussion, the reader may take the help of Figure 3.3. In an ideal schedule, T_i executes continuously in the real interval $\left[\frac{i-1}{wt(T)}, \frac{i}{wt(T)}\right)$ ($\frac{i-1}{wt(T)}$ and $\frac{i}{wt(T)}$ need not be integral) and receives an allocation of a fraction $wt(T)$ of a processor at each instant for a total allocation of 1.0 time unit. Furthermore, the total allocation to T up to time $\frac{i}{wt(T)}$ is i time units. In an actual schedule, T is allocated one processor at each instant it executes. Hence, in an actual schedule, unless T_i commences execution before time $\frac{i}{wt(T)}$, the positive lag constraint in (3.1) will be violated at time $\frac{i}{wt(T)}$. Since scheduling is at slot boundaries in the Pfair model, subtasks commence execution at integral times. Hence, in a Pfair schedule, T_i should commence execution at or before time $\left\lceil \frac{i}{wt(T)} \right\rceil - 1$[6] and complete

[6]Note that the latest non-integral commencement time for T_i is before $\frac{i}{wt(T)}$, and hence, the integral commencement time cannot be specified as $\left\lfloor \frac{i}{wt(T)} \right\rfloor$. This is because if $\frac{i}{wt(T)}$ is an integer, then $\left\lfloor \frac{i}{wt(T)} \right\rfloor = \frac{i}{wt(T)}$.

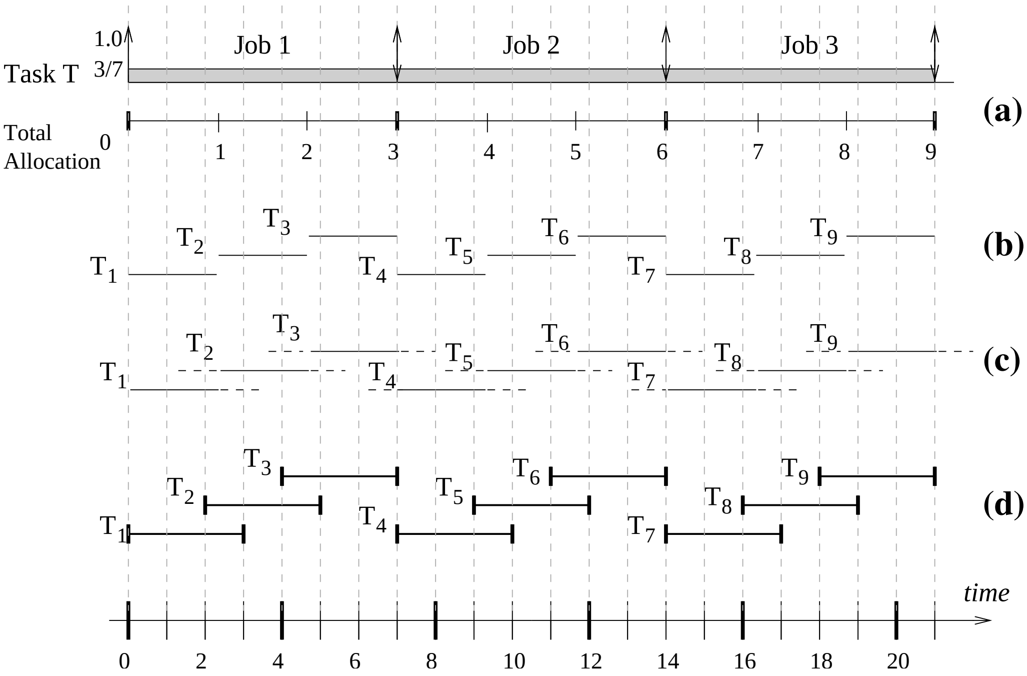


Figure 3.3: **(a)** Allocation to task T with $wt(T) = 3/7$ in an ideal schedule. **(b)** Windows in which subtasks of T execute continuously throughout in an ideal schedule. The end points of these windows need not be at integral times. **(c)** Windows (excluding starting and ending times) within which subtasks must execute for one time unit (and not throughout) in an actual schedule so that constraints on lag specified in (3.1) are not violated. These windows are obtained by extending the windows in inset (b) one time unit to the left and right. The first window is extended only to the right. **(d)** Windows in inset (c) are clipped to slot boundaries. The start time of each window (except the first) in (c) is moved to the first slot boundary that follows the start time; for each window, its end time is moved to the first preceding slot boundary.

by time $\left\lceil \frac{i}{wt(T)} \right\rceil$. Thus, under Pfair scheduling, T_i is said to have a *pseudo-deadline*, denoted $\mathsf{d}(T_i)$, given by

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

Similarly, in an actual schedule, if T_i commences execution at or before time $\frac{i-1}{wt(T)} - 1$ and executes without any interruption, then the negative lag constraint in (3.1) will be violated at time $\frac{i-1}{wt(T)}$. Hence, under Pfair's quantum-based scheduling, if T_i is scheduled in a slot before $\left\lfloor \frac{i-1}{wt(T)} - 1 \right\rfloor + 1 = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$, then (3.1) will not hold. Thus, the *pseudo-release* of T_i, the time before which T_i may not be scheduled, denoted $\mathsf{r}(T_i)$, is given by

$$\mathsf{r}(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor. \tag{3.5}$$

The steps in determining the pseudo-release times and pseudo-deadlines of subtasks is pictorially illustrated in Figure 3.3. The prefix "pseudo," which is used to indicate that the entity under consideration is a subtask and not a job, will henceforth be omitted for brevity.

It can easily be shown that all job deadlines of periodic tasks are met in a Pfair schedule. For this, let $\mathcal{S}$ be a Pfair schedule for a periodic task system τ. The deadline of the k^{th} job of a task T, where $k \geq 1$, is given by $k \cdot T.p$. The total allocation to T up to time $k \cdot T.p$ in an ideal schedule is $k \cdot T.e$. By (R2), $k \cdot T.p$, $(k-1) \cdot T.e$, and $k \cdot T.e$ are integers, and recall that subtasks $(k-1) \cdot T.e + 1, \ldots, k \cdot T.e$ comprise T's k^{th} job. Hence, because a subtask can begin executing only at a slot boundary, subtask $k \cdot T.e$ of T either completes executing by time $k \cdot T.p$ or does not commence execution until $k \cdot T.p$. If the former holds, then the k^{th} job meets its deadline. On the other hand, if the latter holds, then the total allocation to T at $k \cdot T.p$ is at most $k \cdot T.e - 1$. Hence, T's lag at t is at least one, which, by (3.1), contradicts the fact that $\mathcal{S}$ is Pfair.

PF-windows. The interval $[\mathsf{r}(T_i), \mathsf{d}(T_i))$ is referred to as the *Pfair-window* or *PF-window* of T_i and is denoted $\omega(T_i)$. The PF-windows of the first few subtasks of a periodic task with weight 3/7 are shown in Figure 3.4(a). By (3.5) and (3.4), $|\omega(T_i)| = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$. Anderson and Srinivasan have shown the following with respect to the PF-window length of an arbitrary subtask.

Lemma 3.1 (Anderson and Srinivasan [15]) *The length of the PF-window of any subtask T_i of a task T, $|\omega(T_i)| = d(T_i) - r(T_i)$, is either $\left\lceil \frac{1}{wt(T)} \right\rceil$ or $\left\lceil \frac{1}{wt(T)} \right\rceil + 1$.*

***b*-bits.** The *b-bit* or *boundary bit* is associated with each subtask T_i and is denoted $\mathsf{b}(T_i)$. The b-bit is used by some Pfair scheduling algorithms and indicates the number of slots by which the PF-windows of T_i and T_{i+1} overlap. By (3.4), $\mathsf{d}(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil$ holds, and by (3.5), $\mathsf{r}(T_{i+1}) = \left\lfloor \frac{i}{wt(T)} \right\rfloor$. Therefore,

$$\mathsf{b}(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i}{wt(T)} \right\rfloor, \tag{3.6}$$

and hence, $\mathsf{b}(T_i)$ is either zero or one. That is, two consecutive PF-windows are either disjoint or overlap by at most one slot. As will be discussed later, (3.6) also defines the b-bits of subtasks in other task models considered under Pfair scheduling.

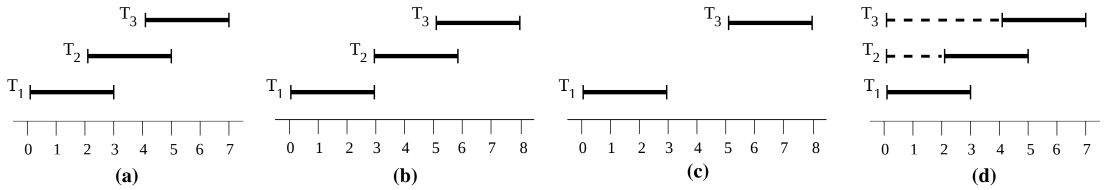


Figure 3.4: **(a)** PF-windows of the first job of a periodic (or sporadic) task T with weight $3/7$. This job consists of subtasks T_1, T_2, and T_3, each of which must be scheduled within its window. (This pattern repeats for every job.) **(b)** PF-windows of an IS task. Subtask T_2 is released one time unit late. Here, $\Theta(T_1) = 0$ while $\Theta(T_2) = \Theta(T_3) = 1$. **(c)** PF-windows of a GIS task. Subtask T_2 is absent and subtask T_3 is released one time unit late. **(d)** PF- and IS-windows of the first job of a GIS task with early releases. All the subtasks of this job are eligible when the job arrives. (The deadline-based priority definition of the Pfair scheduling algorithms and the prohibition of parallel execution of a task ensure that the subtasks execute in the correct sequence.) For each subtask, its PF-window consists of the solid part; the IS-window includes the dashed part, in addition. For example, T_2's PF-window is $[2, 5)$ and its IS-window is $[0, 5)$.

Group deadlines. Like the b-bit, the *group deadline* is a parameter that is associated with each subtask and used by some Pfair scheduling algorithms. The group deadline of subtask T_i is denoted $\mathsf{D}(T_i)$. Group deadlines are used for correctly scheduling task systems with one or more heavy tasks with weights in the range $[1/2, 1)$.

By Lemma 3.1, all the PF-windows of a heavy task with weight less than one are of length two or three. For such a heavy task, "subtask groups," which are maximal sequences of subtasks, satisfying the following two properties with respect to their PF-windows, can be defined: each window, except possibly of the first subtask in the sequence, is of length two, and every consecutive pair of windows is overlapping. In Figure 3.5(a), T_1, T_2 is one such sequence in which the first window is of length two; $T_3, \ldots, T_5$ and $T_6, \ldots, T_8$ are other such sequences with a 3-window[7] as the first window. In each sequence, if any subtask is not scheduled until its last slot, then all subsequent subtasks will be forced to be scheduled in their last slots as well, and in that sense constitute a "group." In addition, if the last subtask in the group is followed by a subtask with a 3-window, as in the first two groups considered above, then this subtask will be precluded from being scheduled in its first slot (when any subtask in the group is scheduled in its last slot). However, no later subtask is directly impacted. Thus, the *group deadline of* T_i can be defined as the earliest time t after $\mathsf{r}(T_i)$ such that t is the release time of some subtask and no subtask released at or after t is directly influenced by whether T_i is scheduled in its last slot.

Informally, for a heavy periodic task with weight less than one, the end of each slot that is

[7]A window spanning k slots is referred to as a *k-window*.

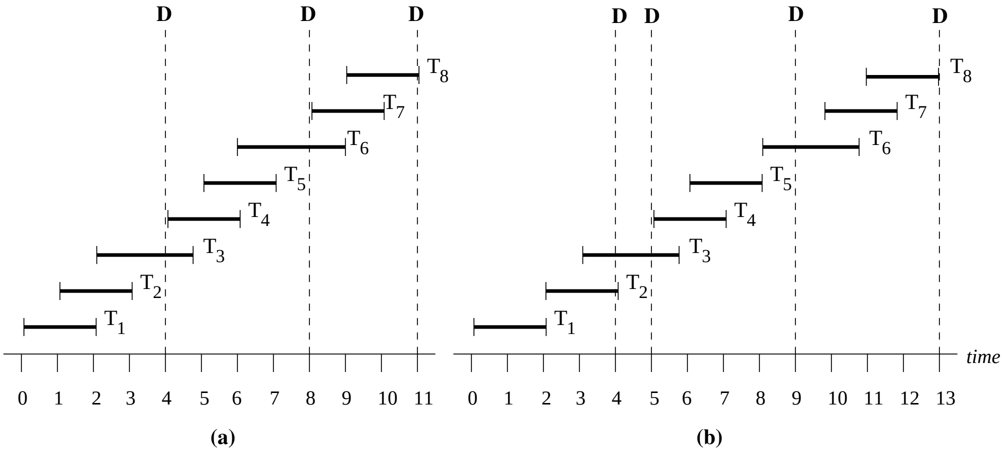


Figure 3.5: Illustration of group deadlines using a task T with weight 8/11. Group deadlines are marked with a "D." **(a)** T is synchronous, periodic. The group deadlines of T_1 and T_2 are at time 4, and those of $T_3, \ldots, T_5$ and $T_6, \ldots, T_8$ are at times 8 and 11, respectively. **(b)** T is an IS task and its subtasks T_2 and T_6 are released late. Nevertheless, the group deadline of T_1 is still at time 4. However, the group deadline of T_2 is at time 5. Similarly, though T_6 is released one time unit late, the group deadlines of $T_3, \ldots, T_5$ are computed under the assumption that T_6 would be released in time, and hence, are at time 9. The group deadlines of $T_6, \ldots, T_8$ are at time 13.

not the first slot of the PF-window of any of its subtasks is a group deadline. In Figure 3.5(a), times 4, 8, and 11 are group deadlines for T in the interval $[0, 11]$. Note that no subtask is released at time 3, 7, or 10. Group deadline of subtask T_i can then be defined as $\mathsf{D}(T_i) = (\min u : u \geq d(T_i) \wedge\ u \text{ is a group deadline of } T)$. For example, in Figure 3.5(a), $\mathsf{D}(T_1) = 4$ and $\mathsf{D}(T_6) = 11$.

Group deadlines of a heavy task T can be computed more directly in constant time by noting that each group deadline of T corresponds to the pseudo-deadline of some subtask of a *complementary task* of T, *i.e.*, a task with weight $1 - wt(T)$. Therefore $\mathsf{D}(T_i)$ is given by the earliest time at or after $\mathsf{d}(T_i)$ that is a pseudo-deadline of a complementary task of T, and can be shown to be $\left\lceil \frac{\left\lceil \frac{i}{wt(T)} \times (1 - wt(T)) \right\rceil}{1 - wt(T)} \right\rceil$, which can be simplified as follows.[8]

$$\mathsf{D}(T_i) = \left\lceil \frac{\left\lceil \frac{i}{wt(T)} \right\rceil - i}{1 - wt(T)} \right\rceil = \left\lceil \frac{\mathsf{d}(T_i) - i}{1 - wt(T)} \right\rceil \tag{3.7}$$

(3.7) has the nice interpretation that the pseudo-deadline we are interested in is given by that

[8]The formula derived in [104] is $\left\lceil \frac{\left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil \times (1 - wt(T)) \right\rceil}{1 - wt(T)} \right\rceil$, which has an extra ceiling that is not needed.

of the $(\mathsf{d}(T_i) - i)^{\text{th}}$ subtask of a complementary task of T.

If T is either light or of weight 1.0, then $\mathsf{D}(T_i)$ is defined to be zero.

3.3 Task Model Extensions

In [25], Baruah *et al.* considered scheduling only synchronous, periodic task systems. Later, Srinivasan and Anderson extended Pfair scheduling to asynchronous, periodic task systems, and some other task models, which are described in this section.

The intra-sporadic task model. The *intra-sporadic* (IS) task model was developed as a means of supporting sporadic task systems [14, 105], and provides a general notion of recurrent execution that subsumes that found in the periodic and sporadic task models. Just as the sporadic model generalizes the periodic model by allowing jobs to be released "late," the IS model generalizes the sporadic model by allowing subtasks to be released late, as illustrated in Figure 3.4(b). More specifically, the separation between $\mathsf{r}(T_i)$ and $\mathsf{r}(T_{i+1})$ is allowed to be more than $\left\lfloor \frac{i}{wt(T)} \right\rfloor - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$, which would be the separation if T were periodic.

The amount of time by which T_i is released late (in comparison to its release time if T were periodic) is referred to as the *offset* of T_i and is denoted $\Theta(T_i)$. If T_i's release is postponed, then its deadline is also postponed by an equal amount. Further, the release of every later subtask is postponed by at least $\Theta(T_i)$, compared to its release when $\Theta(T_i) = 0$. Hence, if a later subtask T_k's release is delayed relative to T_i, then $\Theta(T_k) > \Theta(T_i)$. Therefore, the following holds.

$$k > i \Rightarrow \Theta(T_k) \geq \Theta(T_i) \tag{3.8}$$

Thus, an IS task is obtained by allowing a task's windows to be shifted right from where they would appear if the task were periodic. For a sporadic task, all subtasks that belong to the same job will have equal offsets. By this discussion, and by (3.5) and (3.4), $\mathsf{r}(T_i)$ and $\mathsf{d}(T_i)$ for subtask T_i of an IS task are defined as follows.

$$\mathsf{r}(T_i) = \Theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{3.9}$$

$$\mathsf{d}(T_i) = \Theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil \tag{3.10}$$

The generalized intra-sporadic task model. Apart from the IS task model, Srinivasan and Anderson introduced the *generalized intra-sporadic* (GIS) task model also in [105]. This model extends the IS task model by allowing subtasks to be omitted or skipped; however, the spacing between subtasks that are not skipped may not be decreased in comparison to how they are spaced in a periodic task. Specifically, subtask T_i may be followed by subtask T_k, where $k > i+1$ if the following holds: $\mathsf{r}(T_k) - \mathsf{r}(T_i)$ is at least $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$. That is, $\mathsf{r}(T_k)$ 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, $\mathsf{r}(T_k)$ must be at least $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor$. Figure 3.4(c) shows an example. In this example, though subtask T_2 is omitted, T_3 cannot be released before time 4. 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.

The b-bit and group deadline of a subtask T_i of a GIS or an IS task are computed assuming that all later subtasks are present and are released as early as possible, that is, under the assumption that T_j is released and $\Theta(T_j) = \Theta(T_i)$ holds for all $j \geq i$, regardless of how the subtasks are actually released. Hence, $\mathsf{b}(T_i)$ is given by (3.6), even if T is an IS or a GIS task; $\mathsf{D}(T_i)$ is increased by $\Theta(T_i)$ and is hence given by

$$\mathsf{D}(T_i) = \Theta(T_i) + \left\lceil \frac{\left\lceil \frac{i}{wt(T)} \right\rceil - i}{1 - wt(T)} \right\rceil. \tag{3.11}$$

The method for determining group deadlines for an IS task is illustrated in Figure 3.5(b).

The lag of an IS or a GIS task T at t in schedule $\mathcal{S}$ is also given by (3.1). Computing $\mathsf{A}(\mathcal{S}, T, 0, t)$ is explained towards the end of this section.

The early-release task model. The task models described so far are non-work-conserving in that, to ensure that the lower bound on lag in (3.1) is met, the second and later subtasks of a job remain ineligible to be scheduled before their release times, even if they are otherwise ready and some processor is idle. The *early-release* (ER) task model was introduced by Anderson and Srinivasan in [13] as a work-conserving variant to allow subtasks to be scheduled before their release times. Early releasing can be applied to subtasks in any of the task models considered so far, and unless otherwise specified, it should be assumed that early releasing is enabled. However, whether subtasks are actually released early is optional. To facilitate this, in this model, each subtask T_i has an eligibility time $\mathsf{e}(T_i)$ that specifies the first time slot in which

T_i may be scheduled. It is required that the following hold.

$$(\forall i \geq 1 :: \mathsf{e}(T_i) \leq \mathsf{r}(T_i) \ \wedge \ \mathsf{e}(T_i) \leq \mathsf{e}(T_{i+1})) \tag{3.12}$$

Note that the model is very flexible in the sense that it does not preclude a job from becoming eligible before its release time, but provides mechanisms to restrict such behavior, if so desired. Such flexibility, in conjunction with the sporadic or the IS task model, can be used to schedule rate-based tasks, whose arrival pattern may be jittered, and was first considered by Jeffay and Goddard on uniprocessors in [70].

Because $\mathsf{e}(T_i)$ may be less than $\mathsf{r}(T_i)$, it is possible for an ER task to be ahead of the ideal schedule by an arbitrary amount. Therefore, for such tasks, the negative constraint on lag for Pfairness may not apply, and such tasks are said to be *ERfair*. It should be noted that though an ER subtask may become eligible early, its deadline remains unaltered. The interval $[\mathsf{e}(T_i), \mathsf{d}(T_i))$ is called the *IS-window* of T_i. Figure 3.4(d) shows an example of a task with early releases.

Concrete and non-concrete task systems. As with sporadic task systems, notions of concreteness and non-concreteness can be specified for GIS task systems as well. A GIS task system is said to be *concrete* if release and eligibility times are specified for each subtask of each task, and *non-concrete*, otherwise.

Ideal allocations and lags for GIS task systems. In [14] and [105], Srinivasan and Anderson showed how to compute the ideal allocations in each slot for tasks of a GIS task system (in an ideal schedule), and lags for such tasks in an actual schedule. Since the GIS task model subsumes every other model considered, the formulas are applicable to the other models as well.

The lag of a GIS task T at time t[9] in a schedule $\mathcal{S}$ also is given by (3.2). However, unlike in the periodic case, due to IS separations or omitted subtasks, T may not receive an allocation of $wt(T)$ in every slot in the ideal (PS) schedule, and hence, $\mathsf{A}(\mathsf{PS}, T, 0, t) = wt(T).t$ may not hold. To facilitate expressing $\mathsf{A}(\mathsf{PS}, T, 0, t)$ for GIS tasks, let $\mathsf{A}(\mathsf{PS}, T_i, 0, t)$ and $\mathsf{A}(\mathsf{PS}, T_i, t)$ denote the ideal allocations to subtask T_i in $[0, t)$ and slot t, respectively. We will next discuss how to compute $\mathsf{A}(\mathsf{PS}, T_i, t)$. The reader may take the help of the example in Figure 3.6 in

[9]We remind the reader that all times are integral.

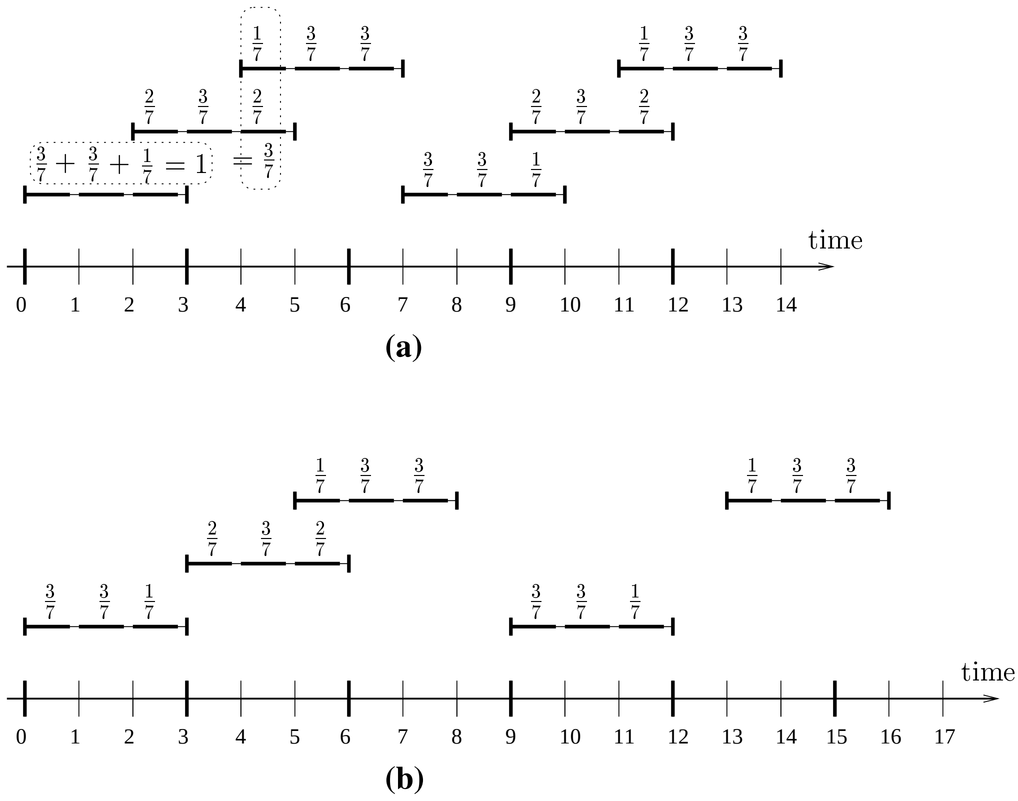


Figure 3.6: Per-slot ideal allocations to subtasks of a task T with weight $3/7$. **(a)** T is periodic: $\mathsf{A}(ideal, T, t) = 3/7$ holds for every t. **(b)** T is GIS: T_2's release is delayed by one time slot; T_4 is delayed by an additional time slot and T_5 is omitted.

following the discussion.

We will begin by assuming that T is periodic. For such tasks, $\mathsf{A}(\mathsf{PS}, T_i, t)$ can be computed from the following: **(i)** the execution requirement of each subtask is one quantum; **(ii)** in an ideal schedule, T receives an allocation of $wt(T)$ in each quantum.

A subtask receives non-zero ideal allocations only in slots that are within its PF-window. Hence, $\mathsf{A}(\mathsf{PS}, T_i, t) = 0$, for $t \notin [\mathsf{r}(T_i), \mathsf{d}(T_i) - 1]$. By (ii), T_i receives an ideal allocation of exactly $wt(T)$ in every slot of its PF-window that does not overlap with that of any other subtask, which includes all slots t, where $\mathsf{r}(T_i) < t < \mathsf{d}(T_i) - 1$. The total ideal allocation to T in slots $0, \ldots, \mathsf{r}(T_i)$, *i.e.*, $\mathsf{A}(\mathsf{PS}, T, 0, \mathsf{r}(T_i) + 1)$, is exactly $wt(T) \cdot (\mathsf{r}(T_i) + 1)$. If $i > 1$, then since the PF-windows of two consecutive subtasks can overlap in at most one slot, the deadline of T_{i-1}, is at or before $\mathsf{r}(T_i) + 1$. Hence, the total ideal allocation in $[0, \mathsf{r}(T_i) + 1)$ to all the subtasks preceding T_i is $i - 1$. If $i = 1$, then it is trivial that the corresponding value is zero, which is equal to $i-1$. It therefore follows that

$$\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) = \mathsf{A}(\mathsf{PS}, T, 0, \mathsf{r}(T_i)+1) - (i-1) = wt(T)\cdot(\mathsf{r}(T_i)+1) - (i-1). \tag{3.13}$$

For example, in Figure 3.6(a), $\mathsf{r}(T_3) = 4$. Hence, $wt(T)\cdot(\mathsf{r}(T_3)+1) = (3/7)\cdot 5 = 15/7$. Here, $i = 3$, and therefore, $\mathsf{A}(\mathsf{PS}, T_3, \mathsf{r}(T_3)) = 15/7 - 2 = 1/7$ as marked in the figure.

To compute $\mathsf{A}(T_i, \mathsf{d}(T_i)-1)$, we note the following. The total ideal allocation to subtasks $T_1, \ldots, T_i$ in $[0, \mathsf{d}(T_i))$ is exactly i quanta. In $[0, \mathsf{d}(T_i)-1)$, T receives a total ideal allocation of $wt(T)\cdot(\mathsf{d}(T_i)-1)$, which is due to subtasks $T_1, \ldots, T_i$ only. Further, no subtask preceding T_i receives a non-zero allocation in slot $\mathsf{d}(T_i)-1$. Therefore,

$$\mathsf{A}(\mathsf{PS}, T_i, \mathsf{d}(T_i)-1) = i - wt(T)\cdot(\mathsf{d}(T_i)-1). \tag{3.14}$$

Referring to the example in Figure 3.6(a) again, $\mathsf{d}(T_2) = 5$. Hence, $wt(T)\cdot(\mathsf{d}(T_2)-1) = (3/7)\cdot 4 = 12/7$. Here $i = 2$, and so, $\mathsf{A}(\mathsf{PS}, T_2, 4) = 2 - 12/7 = 2/7$.

Recall that in an IS task system, each time a subtask release is postponed, the total postponement in the release of each future subtask is increased by an equal amount. Hence, the formulas in (3.13) and (3.14) hold for IS tasks also (with the difference that $\mathsf{r}(T_i)$ and $\mathsf{d}(T_i)$ on the right-hand side correspond to values obtained using (3.5) and (3.4), respectively, *i.e.*, assuming T were periodic, whereas those on the left-hand side correspond to the actual GIS release time and deadline given by (3.9) and (3.10), respectively). Informally, when a subtask is shifted, its per-slot allocations simply move with its window. Finally, allocations to subtasks that are omitted in a GIS task system simply become zero. Per-slot allocations of no other subtasks are impacted. Substituting values for $\mathsf{r}(T_i)$ and $\mathsf{d}(T_i)$ from (3.5) and (3.4) in (3.13) and (3.14) above, $\mathsf{A}(\mathsf{PS}, T_i, t)$ is given by

$$\mathsf{A}(\mathsf{PS}, T_i, u) = \begin{cases} (\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1) \times wt(T) - (i-1), & u = \mathsf{r}(T_i) \\ i - (\left\lceil \frac{i}{wt(T)} \right\rceil - 1) \times wt(T), & u = \mathsf{d}(T_i) - 1 \\ wt(T), & \mathsf{r}(T_i) < u < \mathsf{d}(T_i) - 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.15}$$

As shown in Figure 3.6, $\mathsf{A}(\mathsf{PS}, T, u)$ usually equals $wt(T)$, but in certain slots, it may be less than $wt(T)$ due to omitted or delayed subtasks. Also, the total allocation that a subtask T_i receives in the slots that span its window is exactly one in the ideal schedule. These and similar properties have formally been proved in [104]. Later in this dissertation, we will use Lemmas 3.2 and 3.3, and (3.16)–(3.19) given below (examples of which can be seen in Figure 3.6).

$$(\forall T, u \geq 0 :: \mathsf{A}(\mathsf{PS}, T, u) \leq wt(T)) \tag{3.16}$$

$$(\forall T_i :: \sum_{u=\mathsf{r}(T_i)}^{\mathsf{d}(T_i)-1} \mathsf{A}(\mathsf{PS}, T_i, u) = 1) \tag{3.17}$$

$$(\forall T_i, u \geq 0 :: \mathsf{A}(\mathsf{PS}, T_i, u) \leq wt(T)) \tag{3.18}$$

$$(\forall T_i, u \in [r(T_i), d(T_i)) :: \mathsf{A}(\mathsf{PS}, T_i, u) \geq 1/T.p) \tag{3.19}$$

Lemma 3.2 (A. Srinivasan [104]) *If* $b(T_i) = 1$, *then* $\mathsf{A}(\mathsf{PS}, T_{i+1}, r(T_{i+1})) \leq \rho(T)$, *where* $\rho(T) = \frac{T.e - \gcd(T.e, T.p)}{T.p}$.

Lemma 3.3 (A. Srinivasan [104]) *If* $\mathsf{b}(T_i) = 1$ *and subtask* T_{i+1} *exists, then* $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{d}(T_i) - 1) + \mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{r}(T_{i+1})) = wt(T)$.

A task T's ideal allocation up to time t is simply

$$\mathsf{A}(\mathsf{PS}, T, 0, t) = \sum_{u=0}^{t-1} \mathsf{A}(\mathsf{PS}, T, u) = \sum_{u=0}^{t-1} \sum_{i} \mathsf{A}(\mathsf{PS}, T_i, u),$$

and hence

$$\begin{aligned}
\mathsf{lag}(T, t, \mathcal{S}) &= \mathsf{A}(\mathsf{PS}, T, 0, t) - \mathsf{A}(\mathcal{S}, T, 0, t) && (3.20)\\
&= \sum_{u=0}^{t-1} \mathsf{A}(\mathsf{PS}, T, u) - \sum_{u=0}^{t-1} \mathcal{S}(T, u) && (3.21)\\
&= \sum_{u=0}^{t-1} \sum_{i} \mathsf{A}(\mathsf{PS}, T_i, u) - \sum_{u=0}^{t-1} \mathcal{S}(T, u). && (3.22)
\end{aligned}$$

From (3.21), $\mathsf{lag}(T, t+1)$[10] is given by

$$\begin{aligned}
\mathsf{lag}(T, t+1) &= \sum_{u=0}^{t} (\mathsf{A}(\mathsf{PS}, T, u) - \mathcal{S}(T, u)) \\
&= \mathsf{lag}(T, t) + \mathsf{A}(\mathsf{PS}, T, t) - \mathcal{S}(T, t). && (3.23)
\end{aligned}$$

[10] The schedule parameter is omitted in the lag and LAG functions when unambiguous.

Similarly, by (3.21) again, for any $0 \leq t' \leq t$,

$$\begin{aligned}
\mathsf{lag}(T, t+1) &= \mathsf{lag}(T, t') + \sum_{u=t'}^{t} (\mathsf{A}(\mathsf{PS}, T, u) - \mathcal{S}(T, u)) \\
&= \mathsf{lag}(T, t') + \mathsf{A}(\mathsf{PS}, T, t', t+1) - \mathsf{A}(\mathcal{S}, T, t', t+1) \qquad (3.24) \\
&\leq \mathsf{lag}(T, t') + (t+1-t') \cdot wt(T) - \mathsf{A}(\mathcal{S}, T, t', t+1) \quad \text{(by (3.16))}. \qquad (3.25)
\end{aligned}$$

Another useful definition, the total lag for a task system τ in a schedule $\mathcal{S}$ at time t, $\mathsf{LAG}(\tau, t, \mathcal{S})$, or more concisely, $\mathsf{LAG}(\tau, t)$, is given by

$$\mathsf{LAG}(\tau, t) = \sum_{T \in \tau} \mathsf{lag}(T, t). \qquad (3.26)$$

Using (3.23), (3.24), and (3.26), $\mathsf{LAG}(\tau, t+1)$ can be expressed as follows. In (3.28) below, $0 \leq t' \leq t$ holds.

$$\begin{aligned}
\mathsf{LAG}(\tau, t+1) &= \mathsf{LAG}(\tau, t) + \sum_{T \in \tau} (\mathsf{A}(\mathsf{PS}, T, t) - \mathcal{S}(T, t)) \qquad (3.27) \\
\mathsf{LAG}(\tau, t+1) &= \mathsf{LAG}(\tau, t') + \sum_{u=t'}^{t} \sum_{T \in \tau} (\mathsf{A}(\mathsf{PS}, T, u) - \mathsf{A}(\mathcal{S}, T, u)) \\
&= \mathsf{LAG}(\tau, t') + \mathsf{A}(\mathsf{PS}, \tau, t', t+1) - \mathsf{A}(\mathcal{S}, \tau, t', t+1) \qquad (3.28)
\end{aligned}$$

(3.27) and (3.28) above can be rewritten as follows using (3.16).

$$\begin{aligned}
\mathsf{LAG}(\tau, t+1) &\leq \mathsf{LAG}(\tau, t) + \textstyle\sum_{T \in \tau} (wt(T) - \mathcal{S}(T, t)) \qquad (3.29) \\
\mathsf{LAG}(\tau, t+1) &\leq \mathsf{LAG}(\tau, t') + (t+1-t') \cdot \textstyle\sum_{T \in \tau} wt(T) - \mathsf{A}(\mathcal{S}, \tau, t', t+1) \qquad (3.30) \\
&= \mathsf{LAG}(\tau, t') + (t+1-t') \cdot \textstyle\sum_{T \in \tau} wt(T) - \sum_{u=t'}^{t} \sum_{T \in \tau} \mathcal{S}(T, u) \qquad (3.31)
\end{aligned}$$

3.4 Pfair Scheduling Algorithms

A schedule for a GIS task system is *valid* or *correct* iff each subtask is scheduled in its IS-window. In a valid schedule $\mathcal{S}$, $\mathsf{lag}(T, t, \mathcal{S}) < 1$ holds for all tasks T at all times t. As mentioned above, the constraint $\mathsf{lag}(T, t, \mathcal{S}) > -1$ of (3.1) may not hold due to early releases. As shown in [25] and [14], a valid schedule exists for a GIS task system τ on M processors iff

$$\sum_{T \in \tau} wt(T) \leq M \tag{3.32}$$

holds.

Optimal Pfair algorithms. Pfair scheduling algorithms function by choosing at the beginning of each time slot, at most M eligible subtasks for execution in that time slot. The time complexity of a Pfair scheduling algorithm refers to the worst-case time needed for selecting any such set of M subtasks. At present, three optimal Pfair scheduling algorithms that can correctly schedule any feasible GIS task system in polynomial time are known, namely, PF [25], PD [26], and PD^2 [15, 105].[11] All the three algorithms are based on a common approach: in each algorithm, subtask priorities are first determined on an *earliest-pseudo-deadline-first* basis; ties are then resolved using tie-breaking rules. The algorithms differ in their choice of tie-breaking rules. PF, proposed by Baruah *et al.* in [25], is the earliest, and uses rules that cannot be applied in constant time, and hence, is not considered efficient. PF was followed by PD, which was proposed by Baruah, Gehrke, and Plaxton in [26]. Though tie-breaking under PD requires only constant time, it uses more rules than necessary. In [15], Anderson and Srinivasan showed that two of PD's tie-breaking parameters are redundant, and presented the simpler and more efficient PD^2 algorithm. PF and PD were proved optimal for scheduling only synchronous, periodic task systems. In contrast, Srinivasan and Anderson showed that PD^2 is optimal for scheduling GIS task systems [105]. PD^2's tie-breaking rules subsume those of PF and PD; hence, it follows that PF and PD are also optimal for GIS task systems. As PD^2 is the most efficient of the three algorithms, we limit ourselves to describing the priority definition of PD^2 here. Readers interested in the other algorithms are referred to their primary sources.

Algorithm PD^2. Apart from pseudo-deadlines of subtasks, PD^2 uses their b-bits and group deadlines in determining the priorities of subtasks. 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 hold.

1. $\mathsf{d}(T_i) < \mathsf{d}(U_j)$

[11] PF and PD stand for Pfair and pseudo-deadline, respectively.

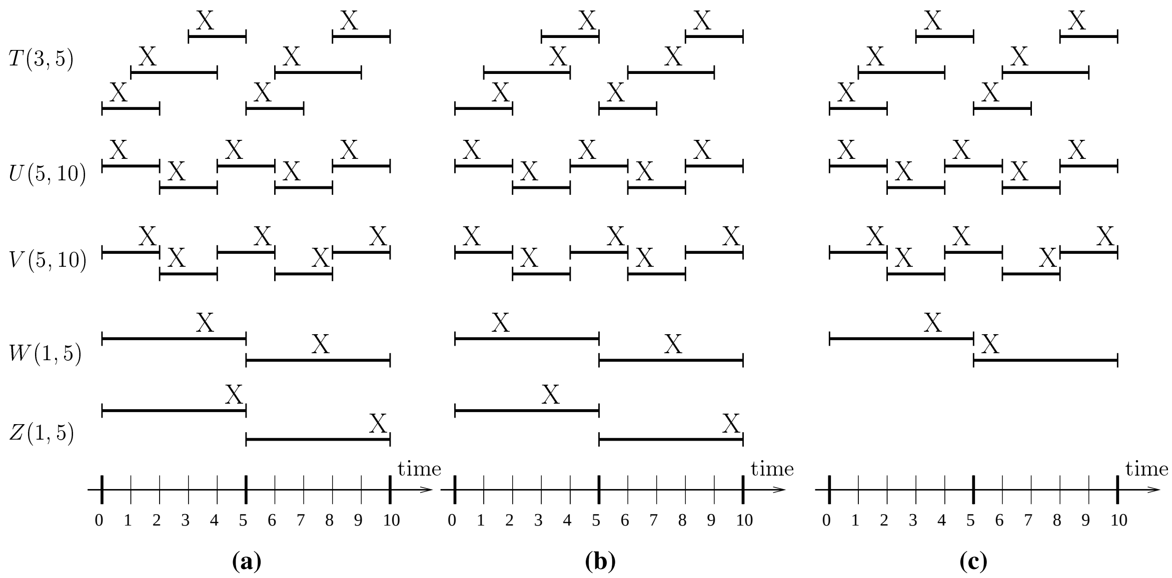


Figure 3.7: A schedule for the first few jobs of tasks T, U, V, W, and Z with weights as specified under **(a)** PD^2 and **(b)** EPDF. **(c)** A schedule for tasks T, U, V, and W under WM. No subtask is early released in any schedule. In each schedule, the slot in which a subtask is scheduled is indicated using an "X."

2. $\mathsf{d}(T_i) = \mathsf{d}(U_j)$ and $\mathsf{b}(T_i) > \mathsf{b}(U_j)$

3. $\mathsf{d}(T_i) = \mathsf{d}(U_j)$, $\mathsf{b}(T_i) = \mathsf{b}(U_j) = 1$, and $\mathsf{D}(T_i) \geq \mathsf{D}(U_j)$.

Any ties that may remain after applying the above rules can be resolved arbitrarily. An example schedule under PD^2 is shown in Figure 3.7(a). The time complexity of PD^2 is $\mathcal{O}(\min(N, M \log N))$.

Algorithm EPDF. EPDF is a derivative of the optimal algorithms in that it schedules subtasks on an earliest-pseudo-deadline-first basis but avoids using any tie-breaking rule. Under EPDF, ties among subtasks with equal deadlines is resolved arbitrarily. Anderson and Srinivasan have shown that, though not optimal in general, EPDF can correctly schedule all task systems that are feasible on two processors [15, 106]. An example schedule under EPDF is shown in Figure 3.7(b). Note the difference in allocations at time 0, in comparison to a schedule under PD^2 for the same task set in inset (a). In [106], Srinivasan and Anderson noted that EPDF may be preferable to the optimal algorithms for soft real-time systems, where bounded tardiness is tolerable, and for dynamic task systems, where the use of tie-breaking rules may not be acceptable. They considered its efficacy for scheduling soft real-time systems, and showed that EPDF can ensure a tardiness of at most q quanta for task systems in which the sum of the weights of the heaviest $M - 1$ tasks is at most $\frac{q \cdot M+1}{q+1}$.

Other algorithms. In the algorithms considered above, task priorities are determined at run-time based on subtask deadlines, and hence, these are dynamic-priority algorithms. More specifically, since the relative priorities between jobs can differ at different times, these are unrestricted-dynamic-priority algorithms. For example, in Figure 3.7, all the subtasks of U and V that are depicted belong to the first jobs of those tasks. At time 0, U's job has a higher priority than V's, whereas at time 1, the converse holds.

Algorithms have been proposed for constructing Pfair schedules using static task priorities also. The motivation for such effort is to enable generating Pfair schedules in systems where practical factors limit the use of dynamic priorities. In [21], Baruah proposed the *weight-monotonic* (WM) scheduling algorithm that prioritizes eligible subtasks on the basis of task weights and developed a sufficient schedulability test for WM on uniprocessor systems. WM was later considered by Moir and Ramamurthy for multiprocessor systems in [96], and generalized by Ramamurthy under the name *rational rate-monotonic* (RRM) for systems with arbitrary deadlines [97].

Despite using static priorities for tasks, algorithms like WM cannot be considered static-priority algorithms in the conventional sense because under Pfair scheduling algorithms, an otherwise ready task may be ineligible for execution so that the negative constraint on lag in (3.1) is not violated. Hence, the relative priorities between two tasks, when considered in the conventional sense, may be different at different times. For example, in the WM schedule in Figure 3.7(c), T has the highest priority when it has an eligible subtask. Each job of T has an execution cost of three, and hence, at time 2, the first job of T is still pending. However, at time 2, T is not scheduled whereas U and V are, and hence, T can be thought of as having a lower priority than these other tasks. Therefore, results on static-priority scheduling discussed in Chapter 1 are not applicable to Pfair algorithms that use static task priorities.

3.5 Practical Enhancements

Subsequent to the initial research on Pfair scheduling, which was described in the previous section, and which is suitable only for independent and static task systems, several mechanisms have been proposed to support real systems wherein tasks may share resources, and task parameters may change at run-time. Techniques have also been proposed to mitigate system overheads and to improve overall resource utilization. Some significant contributions are described in this subsection.

Dynamic task systems. As a first step towards supporting dynamic task systems, Srinivasan and Anderson established conditions under which a task may be allowed to leave or join a GIS task system [107]. A weight change for a task may then be enacted by letting it leave with its old weight and rejoin with a new weight. Block *et al.* have since improved the initial conditions derived, so that quicker and more accurate weight changes are possible [36]. In dynamic systems, spare processing capacity may become available when task parameters change. Distributing this spare capacity among tasks that can consume it in proportion to their weights is a problem of related interest. Schemes based on heuristics have been proposed for this problem [43, 9].

Non-migratory tasks. In [90], Moir and Ramamurthy considered scheduling periodic task systems in which not every task may be migratable but in which some tasks should execute on a specific processor, that is, are "fixed" to that processor.[12] Using a flow graph argument similar to that of Baruah *et al.* [25], they showed that a Pfair schedule exists for such task systems as long as the total weight of the tasks that are fixed to a processor is at most one. They also proposed a hierarchical online algorithm for scheduling such task systems. In their algorithm, all tasks that are fixed to a processor are combined into a single task, called a *supertask*. Each supertask is assigned a weight equal to the sum of the weights of its constituent tasks. A first-level PD^2 scheduler is used for scheduling the supertasks and the migratable tasks. Whenever a supertask is scheduled, a second-level EPDF scheduler is used to schedule one of its constituent non-migratory tasks. This algorithm is illustrated in Figure 3.8. Unfortunately, Moir and Ramamurthy also presented a counterexample (which has been used in Figure 3.8) that showed that this approach is not optimal. Devising an efficient and optimal online algorithm for scheduling task systems with non-migratory tasks still remains an interesting and challenging open problem.

Later, Holman and Anderson studied the supertasking approach in considerable detail. They showed that by *reweighting* a supertask, *i.e.*, assigning it a weight that is slightly higher than the total weight of its constituent tasks, the deadlines of the constituent tasks can be met [65]. They also presented a framework for computing the weight of a supertask based

[12] Different tasks can be fixed to different processors.

on the weights of its constituent tasks, and empirically demonstrated that the overall loss in system utilization due to reweighting should often be reasonable [68].

Synchronization support. In real systems, tasks are not independent, but generally communicate by manipulating data objects that are common to all or a subset of the tasks. Access to shared objects is serialized using synchronization protocols, which may lead to *priority inversions* in real-time systems. A priority inversion is said to occur if a higher-priority task waits for a lower-priority task to relinquish control of some shared object. In multiprocessor systems, shared objects may cause one or more processors to idle, in addition. Hence, if the synchronization protocol is not tailored for multiprocessor real-time systems, then uncontrolled priority inversions and uncontrolled idling may ensue, resulting in an under-utilization of the system resources and making it difficult to provide meaningful real-time guarantees.

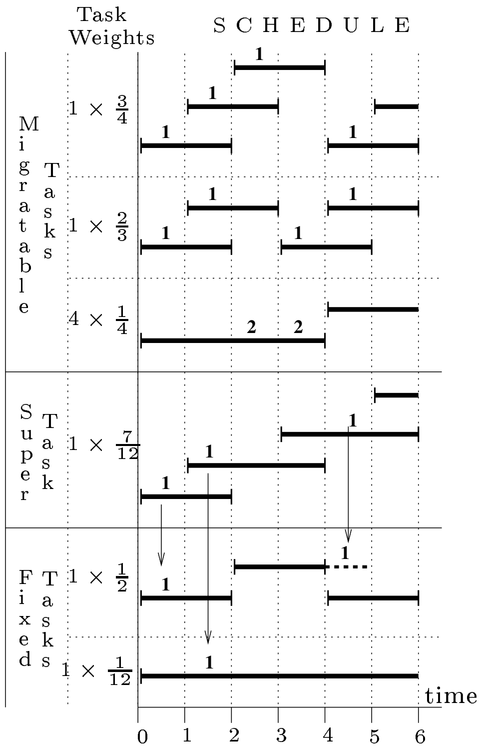


Figure 3.8: A partial schedule based on Mark and Ramamurthy's hierarchical algorithm for a task system composed of four tasks, one each of weight 3/4, 2/3, 1/2, and 1/12, and four other tasks, all of weight 1/4. The tasks with weights 1/2 and 1/12 cannot migrate, and are fixed to a common processor. These two tasks are combined into a supertask with weight 7/12 ($= 1/2 + 1/12$). The supertask and the migratory tasks are scheduled using PD^2 at the first level. The slots in which subtasks are scheduled are indicated by a "1" or "2" over the subtask windows. The four tasks with weight 1/4 are shown as a group. A "2" is marked over the subtask windows of this group only, indicating that two of the four tasks have a subtask scheduled. Allocations to the supertasks are passed on to the fixed tasks using EPDF as the second-level scheduler. In this example, the second subtask of the fixed task with weight 1/2 misses its deadline.

In [67], Holman and Anderson considered the use of *lock-free* algorithms for real-time synchronization in Pfair-scheduled systems. In such algorithms, operations on shared data structures are implemented using "retry loops:" operations are optimistically attempted and retried until successful. (See [12] for an in-depth discussion of lock-free synchronization.) Retries are needed in the event that concurrent operations by different tasks interfere with each other. For lock-free objects to be usable in real-time systems, it is important that bounds on interferences (and hence, retries) be determined.

While the viability of the lock-free approach for *uniprocessor*-based real-time systems is well-known [12], on multiprocessors, lock-free sharing is often considered impractical, because of difficulties in computing reasonable retry bounds. Nevertheless, Holman and Anderson argued and showed that for simple objects, such as queues, stacks, and linked lists, quantum-based scheduling of Pfair algorithms can be exploited to mitigate the overhead considerably. They also proposed combining tasks accessing common objects into a supertask to further reduce overhead.

Holman and Anderson also presented locking synchronization protocols under Pfair algorithms for synchronizing accesses to complex objects, and for use in other scenarios where lock-free approaches may not be suitable [66].

Reducing context-switching costs. In [120], Zhu *et al.* proposed the boundary fair (BF) scheduling algorithm for periodic task systems. This algorithm includes heuristics to reduce the number of scheduler invocations, and thereby, the number of context switches. BF differs from Pfair algorithms in that it makes scheduling decisions only at time instants that are multiples of some task period, called *boundary time points*. At each boundary point t_b, up to $k \cdot M$ subtasks are selected for execution in the interval $[t_b, t_b + k)$, where $t_b + k$ is the next boundary point. A schedule that reduces the number of context switches is then laid out for the selected subtasks.

3.6 Technical Definitions

Active tasks. If subtasks are absent or are released late, then it is possible for a GIS (or IS) task to have no eligible subtasks and an allocation of zero during certain time slots. Tasks with and without subtasks in the interval $[t, t + \ell)$ are distinguished using the following definition of an *active* task.

Definition 3.1: A GIS task U is *active* in slot t if it has one or more subtasks U_j such that $\mathsf{e}(U_j) \leq t < \mathsf{d}(U_j)$. (A task that is active in t is not necessarily scheduled in that slot.)

Holes. If fewer than M tasks are scheduled at time t in $\mathcal{S}$, then one or more processors would be idle in slot t. For each slot, each processor that is idle in that slot is referred to as a *hole*. Hence, if k processors are idle in slot t, then there are said to be k holes in t. The following lemma is a generalization of one proved in [105], and relates an increase in the total lag of τ, LAG, to the presence of holes.

Lemma 3.4 (Srinivasan and Anderson [105]) *If* $\mathsf{LAG}(\tau, t+\ell, \mathcal{S}) > \mathsf{LAG}(\tau, t, \mathcal{S})$, *where* $\ell \geq 1$, *then there is at least one hole in the interval* $[t, t+\ell)$.

Intuitively, if there is no idle processor in slots $t, \ldots, t+\ell-1$, then the total allocation in $\mathcal{S}$ in each of those slots to tasks in τ is equal to M. This is at least the total allocation that τ receives in any slot in the ideal schedule. Therefore, LAG cannot increase.

Task classification(from [105]). Tasks in τ may be classified as follows with respect to a schedule $\mathcal{S}$ and time interval $[t, t+\ell)$.[13]

$A(t, t+\ell)$**:** Set of all tasks that are scheduled in one or more slots in $[t, t+\ell)$.

$B(t, t+\ell)$**:** Set of all tasks that are not scheduled in any slot in $[t, t+\ell)$, but are active in one or more slots in the interval.

$I(t, t+\ell)$**:** Set of all tasks that are neither active nor scheduled in any slot in $[t, t+\ell)$.

As a shorthand, the notation $A(t)$, $B(t)$, and $I(t)$ is used when $\ell = 1$. $A(t, t+\ell)$, $B(t, t+\ell)$, and $I(t, t+\ell)$ form a partition of τ, *i.e.*, the following holds.

$$\left.\begin{array}{l} A(t, t+\ell) \cup B(t, t+\ell) \cup I(t, t+\ell) = \tau \qquad \wedge \\ A(t, t+\ell) \cap B(t, t+\ell) = B(t, t+\ell) \cap I(t, t+\ell) = I(t, t+\ell) \cap A(t, t+\ell) = \emptyset \end{array}\right\} \tag{3.33}$$

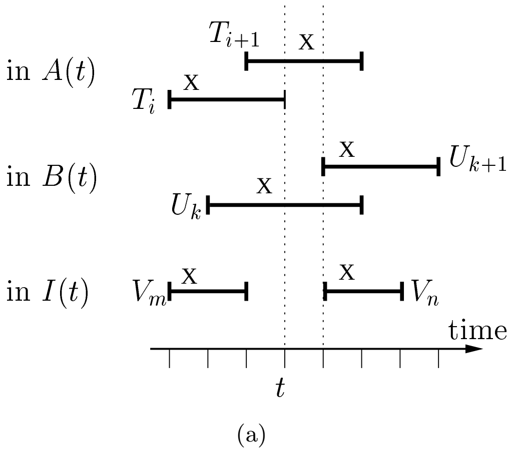
This classification of tasks is illustrated in Figure 3.9(a) for $\ell = 1$. Using (3.26) and (3.33) above, we have the following.

$$\mathsf{LAG}(\tau, t+1) = \sum_{T \in A(t)} \mathsf{lag}(T, t+1) + \sum_{T \in B(t)} \mathsf{lag}(T, t+1) + \sum_{T \in I(t)} \mathsf{lag}(T, t+1) \tag{3.34}$$

The next definition identifies the last-released subtask at t of any task U.

Definition 3.2: Subtask U_j is the *critical subtask of* U *at* t iff $\mathsf{e}(U_j) \leq t < \mathsf{d}(U_j)$ holds, and no other subtask U_k of U, where $k > j$, satisfies $\mathsf{e}(U_k) \leq t < \mathsf{d}(U_k)$.

[13] For brevity, we let the task system τ and schedule $\mathcal{S}$ be implicit in these definitions.



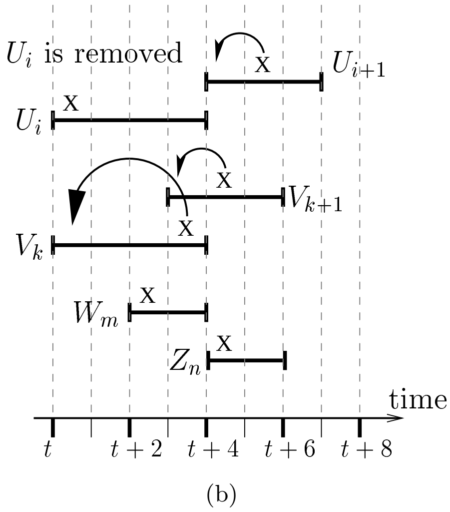


Figure 3.9: **(a)** Illustration of task classification at time t. IS-windows of two consecutive subtasks of three GIS tasks T, U, and V are depicted. The slot in which each subtask is scheduled is indicated by an "X." Because subtask T_{i+1} is scheduled at t, $T \in A(t)$. No subtask of U is scheduled at t; however, because the IS-window of U_k overlaps slot t, U is active at t, and hence, $U \in B(t)$. Task V is neither scheduled at t, nor is it active at t; therefore, $V \in I(t)$. **(b)** Illustration of displacements. If U_i, scheduled at time t, is removed from the task system, then some subtask that is eligible at t, but scheduled later, can be scheduled at t. In this example, it is subtask V_k (scheduled at $t+3$). This displacement of V_k results in two more displacements, those of V_{k+1} and U_{i+1}, as shown. Thus, there are three displacements in all: $\Delta_1 = (U_i, t, V_k, t+3), \Delta_2 = (V_k, t+3, V_{k+1}, t+4)$, and $\Delta_3 = (V_{k+1}, t+4, U_{i+1}, t+5)$.

For example, in Figure 3.9(a), T's critical subtask at both $t-1$ and t is T_{i+1}, and U's critical subtask at $t+1$ is U_{k+1}.

Displacements. To facilitate reasoning about Pfair algorithms, Srinivasan and Anderson formally defined displacements in [105]. Let τ be a GIS task system and let $\mathcal{S}$ be an EPDF schedule for τ. Then, removing a subtask, say T_i, from τ results in another GIS task system τ'. Suppose that T_i is scheduled at t in $\mathcal{S}$. Then, T_i's removal can cause another subtask, say U_j, scheduled after t to shift left to t, which in turn can lead to other shifts, resulting in an EPDF schedule $\mathcal{S}'$ for τ'. Each shift that results due to a subtask removal is called a *displacement* and is denoted by a four-tuple $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$, where $X^{(1)}$ and $X^{(2)}$ represent subtasks. This is equivalent to saying that subtask $X^{(2)}$ originally scheduled at t_2 in $\mathcal{S}$ displaces subtask $X^{(1)}$ scheduled at t_1 in $\mathcal{S}$. A displacement $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ is *valid* iff $\mathsf{e}(X^{(2)}) \leq t_1$. Because there can be a cascade of shifts, there may be a chain of displacements. Such a chain is represented by a sequence of four-tuples. An example is given in Figure 3.9(b).

The next two lemmas regarding displacements are proved in [105] and [104]. The first lemma states that in an EPDF schedule, a subtask removal can cause other subtasks to shift only to their left. According to the second lemma, if a subtask displaces to a slot with a hole, then its predecessor is scheduled in that slot prior to the displacement.

Lemma 3.5 (from [104]) *Let $X^{(1)}$ be a subtask that is removed from τ, and let the resulting chain of displacements in an* EPDF *schedule for τ 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 $i \in [1, k]$.*

Lemma 3.6 (from [105]) *Let $\Delta = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \rangle$ be a valid displacement in any* EPDF *schedule. If $t_i < t_{i+1}$ holds and there is a hole in slot t_i in that schedule, then $X^{(i+1)}$ is the successor of $X^{(i)}$.*

3.7 Summary

In this chapter, we first motivated the concept of Pfair scheduling and described the Pfair scheduling of synchronous, periodic tasks. We then considered some other more flexible recurrent task models that can be Pfair scheduled. Later, we discussed some Pfair scheduling algorithms, and described some techniques that have been proposed to enhance the practicality of Pfair scheduling. Finally, we summarized some technical definitions and results from prior work that will be used in the analyses presented in Chapters 6 and 7. In these chapters, we consider the non-optimal EPDF scheduling algorithm in detail, and present some new results concerning this algorithm. As noted in this chapter, EPDF may be preferable to optimal algorithms for scheduling soft or dynamic real-time systems.

Chapter 4

Tardiness Bounds under Preemptive and Non-Preemptive Global EDF[1]

In this chapter, we derive tardiness bounds that can be guaranteed under preemptive and non-preemptive global EDF (g-EDF and g-NP-EDF) for sporadic real-time task systems on multiprocessors. In general, overheads due to task preemptions and migrations are lower under EDF than under the known optimal algorithms. Further, since EDF does not require the restrictions, which were discussed in Section 1.6, imposed by optimal algorithms, EDF may be preferable to the optimal algorithms in some settings. However, the worst-case schedulable utilizations of g-EDF and g-NP-EDF can be quite low, and hence, using their existing validation tests, which are designed to ensure that no deadline is missed, can be overkill for scheduling soft real-time systems. The tardiness bounds that we determine in this chapter can instead be used for validating such systems. As will be seen, no restriction on total system utilization, except that it not exceed the available processing capacity, is necessary to ensure bounded tardiness under g-EDF and g-NP-EDF. Hence, effective utilization on multiprocessors can be improved by using these results while scheduling soft real-time systems.

Instead of deriving tardiness bounds independently for g-EDF and g-NP-EDF, we consider a more general task model, in which tasks may consist of both preemptive and non-preemptive segments, and a mixed mode EDF scheduler that allows/restricts preemptions based on the nature of the task segment under execution. Such a mixed-mode model may be useful for

[1]Contents of this chapter previously appeared in preliminary form in the following paper:
[51] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, December 2005.

modeling and analyzing some non-independent tasks, such as tasks that share short critical sections [53]. Bounds for g-EDF and g-NP-EDF are deduced as special cases of that derived for the general case. Furthermore, the tardiness bounds derived apply for a variation of the sporadic task model in which jobs may be "early released," *i.e.*, become eligible for execution before their designated release times. (This is similar to the early releasing of subtasks under Pfair scheduling, described in Chapter 3.)

This chapter is organized as follows. It begins with a discussion of some relevant prior work on global scheduling algorithms in Section 4.1. The mixed preemptive/non-preemptive task model and the *early-release* (ER) variation mentioned above, and some notation that is needed in addition to that provided in Chapter 1, are formally presented in Section 4.2. This is followed by a derivation of a tardiness bound for mixed-mode scheduling, and hence, those for g-EDF and g-NP-EDF, in Section 4.3. Then, in Section 4.4, an extension is proposed to the ER variant of the sporadic task model and the issue of determining tardiness bounds for the extended model is discussed. Section 4.5 presents a simulation-based evaluation of the tightness of the bounds (for the basic model). Section 4.6 concludes.

4.1 Global Scheduling

As mentioned in Chapter 1, two primary approaches traditionally considered for scheduling on multiprocessors are partitioning and global scheduling. These approaches were described in detail in Section 1.4.2.2 of that chapter. Of the two approaches, traditionally, partitioning has been preferred perhaps for the following reasons: **(i)** partitioning eliminates overheads due to task migrations; **(ii)** partitioning is conceptually simpler and analytically more tractable; **(iii)** schedulable utilization achieved in practice is higher under partitioning.

Despite the above, some recent developments in computer architecture, as discussed in Chapter 1, and scheduling research provide reasons, which are listed below, to believe that global scheduling may be viable in many settings, and the the two approaches must be reassessed. First, task migrations may be of less concern in modern systems with high-speed processor-to-memory interconnects, in architectures with shared caches, such as some multicore designs, or in embedded systems with no cache. Second, as discussed below, significant progress has been made in the development of techniques for reasoning about global scheduling algorithms and in understanding their behavior. Finally, as briefly mentioned in Chapter 1, global scheduling may be superior to partitioning for soft real-time systems in that partitioning

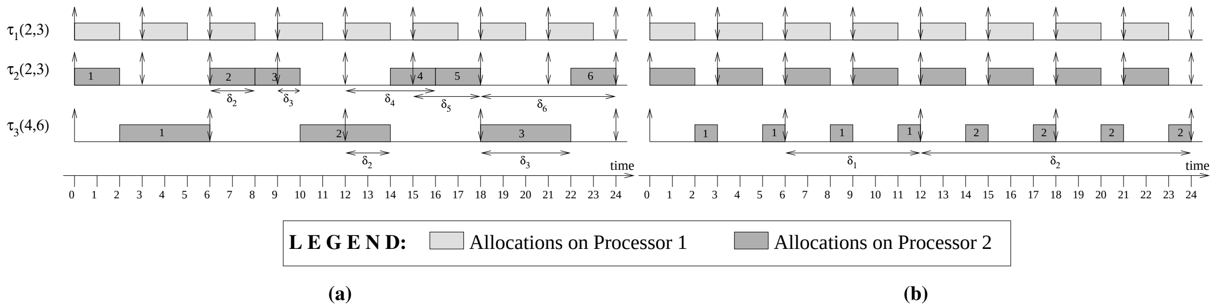


Figure 4.1: A schedule for the task system with parameters as indicated under **(a)** partitioned EDF (with task τ_1 assigned to processor 1 and tasks τ_2 and τ_3 assigned to processor 2), and **(b)** global RM. Numbers within shaded rectangles indicate job numbers. δ_i indicates the tardiness of the i^{th} job of the corresponding task.

schemes offer no scope for improving system utilization even if bounded tardiness can be tolerated. This is because if a task set cannot be partitioned without over-utilizing some processor, then deadline misses and tardiness for tasks on that processor will increase with time. Such an example under partitioned EDF is provided in Figure 4.1(a). In this example, τ_1 is assigned to Processor 1 and τ_2 and τ_3 are assigned to Processor 2. Assume that each job of every task is released as early as permissible and that deadline ties between τ_2 and τ_3 are resolved in favor of τ_3. Here, the second job of τ_3 does not complete executing until time 14, for a tardiness of 2 time units, and the third job incurs a tardiness of 4 time units. It is easy to see that the $(i+1)^{\text{st}}$ job of τ_3 does not complete until time $8i+6$, for all $i \geq 1$, for a tardiness of $2i$ time units. This tardiness increases with time and thus is unbounded. Similarly, tardiness is unbounded for τ_2 as well.

Prior work on global scheduling. One factor that has led many researchers to view global scheduling negatively is the so-called "Dhall effect," which was reported by Dhall and Liu [54] as early as 1978. Dhall and Liu showed that for every $M \geq 2$, there exist task systems with total utilization arbitrarily close to 1.0 that cannot be correctly scheduled on M processors under global EDF or RM scheduling [54]. However, recently, several researchers have noted that the Dhall effect is due to the presence of tasks with high and low utilizations and have shown that it can be overcome by restricting per-task utilizations, and by according higher priorities to high-utilization tasks [109, 61, 22, 20]. In [61], Goossens, Funk, and Baruah showed that on M processors, EDF can correctly schedule any independent periodic task system (with implicit deadlines) if the total utilization of all tasks, U_{sum}, is at most $M-(M-1)u_{\max}$, where $u_{\max}$ is the maximum utilization of any task. Later, Baruah proposed a variant of

EDF that prioritizes tasks with utilization exceeding 1/2 over the rest, and showed that the modified algorithm can correctly schedule any task system with total utilization not exceeding $(M+1)/2$. Schedulability tests that can be applied to task systems in which relative deadlines of tasks can be less than their periods (*i.e.*, constrained-deadline systems) have been developed by Baker [20] and by Bertogna *et al.* [33].

The proposed schedulability tests for implicit-deadline systems that depend on $u_{\max}$ have the property that, the total utilization of a task system that is schedulable increases as $u_{\max}$ is decreased. Nevertheless, even with $u_{\max} = 0.5$, half the total available processing capacity will have to be wasted, if every deadline must be met. This may be overkill for soft real-time systems that can tolerate bounded deadline misses.

The research discussed above is for preemptive global EDF, or g-EDF. To our knowledge, non-preemptive global EDF (g-NP-EDF) has been considered only in [23], where a sufficient schedulability condition is derived for task systems in which the maximum execution cost of any task is less than the minimum period of any task.

Non-trivial schedulability tests have been developed for global RM scheduling also [16, 20, 34]. However, the schedulable utilizations that these tests allow are less than that allowed by g-EDF. Furthermore, like partitioning algorithms, global RM (or any global static-priority algorithm) may not be suitable for soft real-time systems. This is because, task systems exist in which tardiness for low-priority tasks increases with time when scheduled under RM. Refer to Figure 4.1(b) for an example. In this example, the i^{th} job of τ_3 does not complete until time $12i$, for all i, for a tardiness of $6i$ time units.

With the above overview of the state-of-the-art in global scheduling, we turn to deriving tardiness bounds for global EDF. We begin by presenting the additional elements of the task model used in this chapter.

4.2 Task Model and Notation

The basic task model is the sporadic task model described in Section 1.3.1, augmented for soft real-time systems as described in Section 1.5. In this section, we describe two generalizations of the basic task model that we consider in this chapter and also present some additional notation.

We would like to remind the reader that N denotes the number of tasks in the task system

τ under consideration, and M, the number of processors. The total utilization of all the tasks in τ, $U_{sum}(\tau)$, is assumed to not exceed M.

Non-preemptive sections. To provide a common analytic framework for g-EDF and g-NP-EDF, we consider a more general model than the sporadic task model wherein each job of each task τ_i may consist of one or more preemptive and non-preemptive code segments. The maximum execution cost of any non-preemptive segment of τ_i is denoted b_i and the maximum value taken over all tasks in τ is denoted $b_{\max}(\tau)$. In this extended model, task τ_i is denoted using a triple (e_i, p_i, b_i).

Mixed preemptive/non-preemptive scheduling. We will refer to the EDF algorithm that is cognizant of the non-preemptive segments of a task and executes them non-preemptively as EDF-P-NP. (In a real implementation, special system calls would be used to inform the scheduler when a non-preemptive segment is entered and exited.) At any time, higher priority is accorded to jobs with earlier absolute deadlines, subject to not preempting a job that is executing in a non-preemptive segment. Ties are resolved arbitrarily but consistently in that ties between two jobs are resolved identically at all times. A job executing in a preemptive segment may be preempted by an arriving higher-priority job and may later resume execution on the same or a different processor. EDF-P-NP reduces to g-EDF when $b_{\max} = 0$ and to g-NP-EDF when $b_i = e_i$, for all i.

Early releasing. In the model considered in this chapter, jobs of sporadic tasks may be executed "early," *i.e.*, before their release times (as stipulated by the sporadic task model), provided that prior jobs of the same task have completed execution. This is just an application of the concept described in the context of Pfair scheduling of subtasks in Chapter 3 to task systems in which jobs are first-class objects. As mentioned in Chapter 3, the flexibility offered by early releasing can be used for scheduling rate-based tasks [70].

A job that may be executed before its release time is said to be *early released*. Each job $\tau_{i,j}$ is associated with an *eligibility time*, denoted $\ell_{i,j}$, where $\ell_{i,j} \leq r_{i,j}$ and $\ell_{i,j+1} \geq \ell_{i,j}$ for all i and j. $\ell_{i,j}$ is the earliest time that $\tau_{i,j}$ can commence execution. Early releasing does not alter job deadlines, and hence, how job priorities are determined, but only the set of jobs that can contend at a given time. When early releasing is enabled, for a concrete task system, apart from the release time and actual execution cost, the eligibility time of each job is also specified.

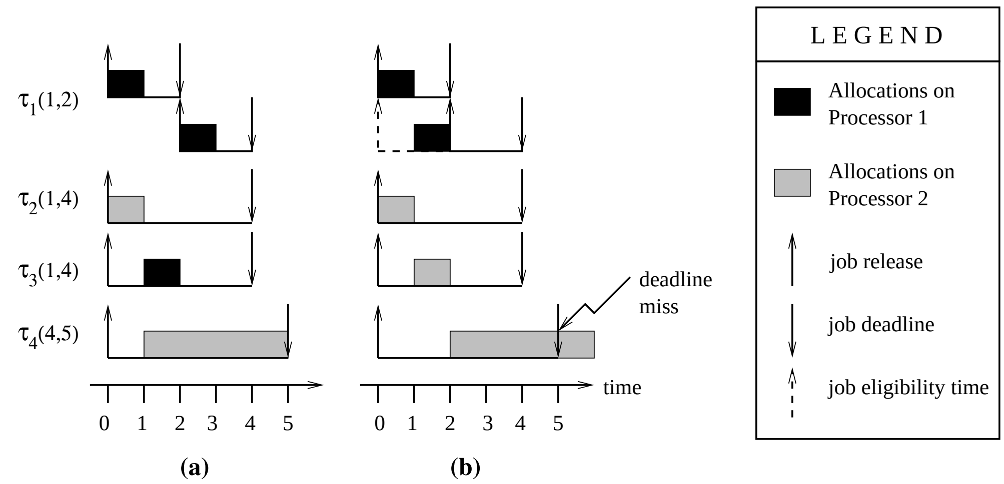


Figure 4.2: Illustration of deadline misses due to early releasing. EDF schedules are shown for some initial jobs of a task system with four tasks (as indicated) and $U_{sum} = 1.8$ on two processors. **(a)** No job is released early. **(b)** Job $\tau_{1,2}$ is early released at time 0. Scheduling $\tau_{1,2}$ before its release time (at time 1) leads to a deadline miss.

On uniprocessors, a swapping argument (that can be used to establish the optimality of EDF) can be used to show that allowing early releases cannot lead to deadline misses for a task system (or a set of jobs) that is otherwise schedulable by EDF. However, the same does not hold for EDF on multiprocessors. An example of a task system in which deadlines are missed due to early releasing is shown in Figure 4.2.[2] Nevertheless, we show that the tardiness bounds that we derive in this chapter hold even if jobs are early released.

Additional notation. The maximum utilization and the maximum and minimum execution cost of any task are denoted $u_{\max}(\tau)$, $e_{\max}(\tau)$, and $e_{\min}(\tau)$, respectively. The task system τ may be omitted from this notation when unambiguous.

The tardiness bound we derive is expressed in terms of the highest task execution costs, utilizations, and non-preemptive segment costs, and the total system utilization. To facilitate expressing the bound, we define the following: ϵ_i (resp., μ_i) denotes the i^{th} execution cost (resp., task utilization) in non-increasing order of the execution costs (resp., utilizations) of all the tasks. (Note that ϵ_i and μ_i need not be parameters of the same task for any i.) Similarly, β_i

[2]The task system in this figure is not guaranteed to be schedulable under g-EDF by any of the known g-EDF sufficient schedulability tests [61, 20, 33] We believe that deadlines will not be missed even with early releasing if a task system satisfies at least one of these tests, and in Appendix B, show that this property holds for a task system satisfying the test in [61].

denotes the i^{th} largest non-preemptive segment cost. For simplicity, we assume the following.

$$(\forall i, j \leq N :: e_i \leq e_j \Leftrightarrow b_i \leq b_j) \tag{4.1}$$

The above assumption can easily be seen to hold for both g-EDF and g-NP-EDF. It can be eliminated for mixed-mode tasks at the expense of either a slightly higher tardiness bound or a more complicated algorithm (as opposed to a closed-form expression) for choosing tasks whose execution costs are to be used in computing a less-pessimistic bound. One such algorithm is described in Section A.3 of Appendix A.

We define the following subsets of tasks in order to lower the pessimism in the bounds specified. In these definitions, $k \leq N$ and $k_2 \leq k_1 \leq N$ hold. (These definitions may be skipped on first reading without loss of continuity and referred back to when needed.) The purpose of defining these sets is follows. The tardiness bounds derived are dependent upon the sum of the execution costs of tasks in some subset of all the tasks and that of the non-preemptive segment costs of tasks in another disjoint subset. While it is simpler to use the maximum execution costs and maximum non-preemptive segment costs as upper bounds, that approach overlooks the fact that the subsets are disjoint, and hence, that at most one of the execution cost and non-preemptive segment cost of a task may be used. The challenge is to identify tasks for the two subsets that can be used as an upper bound for the needed quantity. If (4.1) holds, then $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$, as defined below, serve the purpose.[3] The issue of choosing tasks for these subsets when (4.1) does not hold is discussed in Section A.3.

$$\begin{aligned}
\Gamma^{(k)} &\stackrel{\text{def}}{=} \text{Subset of } k \text{ tasks of } \tau \text{ with the highest execution costs} && (4.2)\\
\Gamma^{(k_1,k_2)} &\stackrel{\text{def}}{=} \text{Subset of } k_2 \text{ tasks } \tau_i \text{ of } \Gamma^{(k_1)} \text{ with the highest values for } e_i - b_i && (4.3)\\
\Pi^{(k_1,k_2)} &\stackrel{\text{def}}{=} \Gamma^{(k_1)} \setminus \Gamma^{(k_1,k_2)} && (4.4)
\end{aligned}$$

Ties in selecting tasks for $\Gamma^{(k)}$ are resolved using task indices. When choosing tasks for $\Gamma^{(k_1,k_2)}$ ties are first resolved in favor of tasks with higher execution costs; any remaining ties are

[3]If (4.1) does not hold, then these definitions are not applicable. In such a case, $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$ will have to be determined iteratively. In either case, defining $\Gamma^{(k_1,k_2)}$ (resp., $\Pi^{(k_1,k_2)}$) as a subset of k_1 (resp., $k_2 - k_1$) tasks of τ with the highest execution costs (resp., non-preemptive segment costs) will serve as a looser upper bound.

i	1	2	3	4	5	6	7	8	9
e_i	20	10	16	2	15	4	12	4	20
p_i	60	15	20	10	20	16	20	10	40
u_i	0.33	0.67	0.8	0.2	0.75	0.25	0.6	0.4	0.5
b_i	7	2	6	0	3	0.5	1	0	7
ϵ_i	20	20	16	15	12	10	4	4	2
μ_i	0.8	0.75	0.67	0.6	0.5	0.4	0.33	0.25	0.2
β_i	7	7	6	3	2	1	0.5	0	0

Table 4.1: Illustration of notation.

resolved using task indices. The definitions above imply the following.

$$\Gamma^{(k_1,k_2)} \cup \Pi^{(k_1,k_2)} = \Gamma^{(k_1)} \tag{4.5}$$
$$\Gamma^{(k_1,k_2)} \cap \Pi^{(k_1,k_2)} = \emptyset$$

Finally, Λ is defined as follows.

$$\Lambda = \begin{cases} U_{sum}(\tau) - 1, & U_{sum}(\tau) \text{ is integral} \\ \lfloor U_{sum}(\tau) \rfloor, & \text{otherwise} \end{cases} \tag{4.6}$$

Example 4.1. Let τ be a task system with nine tasks τ_1 through τ_9 as follows: $\tau = \{\tau_1(20, 60, 7),\ \tau_2(10, 15, 2),\ \tau_3(16, 20, 6),\ \tau_4(2, 10, 0),\ \tau_5(15, 20, 3),\ \tau_6(4, 16, 0.5),\ \tau_7(12, 20, 1),\ \tau_8(4, 10, 0),\ \tau_9(20, 40, 7)\}$. In this example, $U_{sum} = 4.5$, $u_{\max} = u_3 = 0.8$, $e_{\max} = e_1 = e_9 = 20$, $e_{\min} = e_4 = 2.0$, and $b_{\max} = b_1 = b_9 = 7$. ϵ and μ values for this task set are enumerated in Table 4.1. For this task set, $\Gamma^{(5)} = \{\tau_1, \tau_9, \tau_3, \tau_5, \tau_7\}$. Since $e_1 - b_1 = e_9 - b_9 > e_5 - b_5 > e_7 - b_7 > e_3 - b_3$, we have $\Gamma^{(5,3)} = \{\tau_1, \tau_9, \tau_5\}$ and $\Pi^{(5,3)} = \{\tau_7, \tau_3\}$.

4.3 A Tardiness Bound under EDF-P-NP

In this section, we derive a tardiness bound under EDF-P-NP. Our approach involves comparing the allocations to a concrete task system τ in a processor sharing (PS) schedule for τ, which is used as a reference system, and an actual EDF-P-NP schedule of interest for τ, both on M processors, and quantifying the difference between the two. As described in Chapter 3, a PS schedule is an ideal fluid schedule in which each task executes at a precisely uniform rate given by its utilization. Similar proof techniques that use a fluid schedule for reference have previously been considered in research on fair bandwidth allocation in computer networks [94] and fair uniprocessor [114] and multiprocessor [25, 105] scheduling algorithms. In particular, our derivation borrows techniques developed by Srinivasan and Anderson for reasoning about Pfair algorithms on multiprocessors [105, 108, 107, 106]. Pfair scheduling is described in greater detail in Chapter 3 and considered in later chapters. The applicability to EDF of techniques used with fair scheduling algorithms is not surprising since fair schedulers also use some variant of deadline-based scheduling, and EDF also approximates an ideal fluid scheduler, although more coarsely than fair schedulers. Valente and Lipari also use a similar approach in deriving a tardiness bound under g-EDF [116].[4] In Appendix B, we show that one of the known schedulability tests for EDF, namely, $U_{sum} \leq M - (M-1) \cdot u_{\max}$ can also be derived using similar techniques, even when jobs may be early released.

Because $D_i = p_i$, for all i, and $U_{sum} \leq M$ holds, the total demand at any instant will not exceed M in a PS schedule, and hence no deadlines will be missed; in fact, every job will complete executing exactly at its deadline. We begin by setting the required machinery in place.

4.3.1 Definitions and Notation

All times referred to in this chapter are real. The system start time is assumed to be zero. The time interval $[t_1, t_2)$, where $t_2 \geq t_1$, consists of all times t, where $t_1 \leq t < t_2$, and is of length $t_2 - t_1$. The interval (t_1, t_2) excludes t_1. For any time $t > 0$, the notation t^- is used to denote the time $t - \epsilon$ in the limit $\epsilon \to 0+$.

Definition 4.1 (active tasks, active jobs, and windows): A task τ_i is said to be *active* at time t if there exists a job $\tau_{i,j}$ (called τ_i's *active job* at t) such that $r_{i,j} \leq t < d_{i,j}$ (*i.e.*, $\tau_{i,j}$'s release time has elapsed, but its deadline has not). The interval $[r_{i,j}, d_{i,j})$ is referred to as the *window* of $\tau_{i,j}$. By our task model, $D_i = p_i$ holds, hence every task can have at most one active job at any time. $\tau_{i,j}$ is not considered to be active before its release time even if its eligibility time, $\ell_{i,j}$, is earlier.

[4] As publicly reported by the first author of [116] at RTSS '05, the proof published in [116] for the tardiness bound claimed is in error. To our knowledge, an updated result has not yet been published at a refereed venue.

Definition 4.2 (pending jobs): $\tau_{i,j}$ is said to be *pending* at t in a schedule $\mathcal{S}$ if $r_{i,j} \le t$ and $\tau_{i,j}$ has not completed execution by t in $\mathcal{S}$. Note that a job with a deadline at or before t is not considered to be active at t even if it is pending at t. Also, $\tau_{i,j}$ is not considered pending before its release time even if its eligibility time is earlier.

Definition 4.3 (ready jobs): $\tau_{i,j}$ is said to be *ready* at time t in a schedule $\mathcal{S}$ if $t \ge \ell_{i,j}$, and all prior jobs of τ_i have completed execution by t in $\mathcal{S}$, but $\tau_{i,j}$ has not.

Because we compare allocations in a PS schedule and an actual schedule in our analysis, we use the notion of lag described in Chapter 3. However, since time is discrete under Pfair scheduling, whereas it is continuous here, we provide formulas for lag and LAG for the continuous case. Though some are identical to the formulas for the discrete case, we state them, nevertheless, for ease of reference.

As in Chapter 3, let $\mathsf{A}(\mathcal{S}, \tau_i, t_1, t_2)$ denote the total time allocated to τ_i in an arbitrary schedule $\mathcal{S}$ for τ in $[t_1, t_2)$. In PS_τ, τ_i is allocated a fraction u_i of each instant at which it is active in $[t_1, t_2)$, regardless of whether its active job is executing in a preemptive or a non-preemptive section. Also, non-preemptivity constraints are ignored within PS schedules. Hence,

$$\mathsf{A}(\mathsf{PS}_\tau, \tau_i, t_1, t_2) \le (t_2 - t_1) \cdot u_i. \tag{4.7}$$

The total allocation to τ in the same interval in PS_τ is then given by

$$\mathsf{A}(\mathsf{PS}_\tau, \tau, t_1, t_2) \le \sum_{\tau_i \in \tau} (t_2 - t_1) u_i = U_{sum} \cdot (t_2 - t_1) \le M \cdot (t_2 - t_1). \tag{4.8}$$

Hence,

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\mathsf{PS}_\tau, \tau_i, 0, t) - \mathsf{A}(\mathcal{S}, \tau_i, 0, t). \tag{4.9}$$

Schedule $\mathcal{S}$ has performed less work on the jobs of τ_i until t than PS_τ if $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is positive (or τ_i is under-allocated in $\mathcal{S}$), and more work if $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is negative (or τ_i is over-allocated in $\mathcal{S}$). The total lag of a task system τ at t, denoted $\mathsf{LAG}(\tau, t, \mathcal{S})$, is given by the following.

$$\begin{aligned} \mathsf{LAG}(\tau, t, \mathcal{S}) &= \mathsf{A}(\mathsf{PS}_\tau, \tau, 0, t) - \mathsf{A}(\mathcal{S}, \tau, 0, t) \\ &= \sum_{\tau_i \in \tau} \mathsf{A}(\mathsf{PS}_\tau, \tau_i, 0, t) - \sum_{\tau_i \in \tau} \mathsf{A}(\mathcal{S}, \tau_i, 0, t) \end{aligned}$$

$$= \sum_{\tau_i \in \tau} \mathsf{lag}(\tau_i, t, \mathcal{S}) \tag{4.10}$$

$\mathsf{LAG}(\tau, 0, \mathcal{S})$ and $\mathsf{lag}(\tau_i, 0, \mathcal{S})$ are both zero, and by (4.9) and (4.10), we have the following for $t_2 > t_1$.

$$\begin{aligned}
\mathsf{lag}(\tau_i, t_2, \mathcal{S}) &= \mathsf{lag}(\tau_i, t_1, \mathcal{S}) + \mathsf{A}(\mathsf{PS}_\tau, \tau_i, t_1, t_2) - \mathsf{A}(\mathcal{S}, \tau_i, t_1, t_2) && (4.11)\\
\mathsf{LAG}(\tau, t_2, \mathcal{S}) &= \mathsf{LAG}(\tau, t_1, \mathcal{S}) + \mathsf{A}(\mathsf{PS}_\tau, \tau, t_1, t_2) - \mathsf{A}(\mathcal{S}, \tau, t_1, t_2) && (4.12)
\end{aligned}$$

Lag for jobs. The notion of lag defined above for tasks and task sets can be applied to jobs and job sets in an obvious manner. Let τ denote a concrete task system, and Ψ a subset of all jobs in τ with deadlines not later than a specified time. Let $\mathsf{A}(\mathsf{PS}_\tau, \tau_{i,j}, t_1, t_2)$ and $\mathsf{A}(\mathcal{S}, \tau_{i,j}, t_1, t_2)$ denote the allocations to $\tau_{i,j}$ in $[t_1, t_2)$ in PS_τ and $\mathcal{S}$, respectively. Then, $\mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) = \mathsf{A}(\mathsf{PS}_\tau, \tau_{i,j}, r_{i,j}, t) - \mathsf{A}(\mathcal{S}, \tau_{i,j}, \ell_{i,j}, t)$. (This is because a job cannot receive any allocation outside its window in a PS schedule, and before its eligibility time in an actual schedule.) The total allocation in $\mathcal{S}$ in $[0, t)$, where $t > 0$, to a job that is not pending at t^- in $\mathcal{S}$ is at least its allocation in the same interval in PS_τ. Hence, lag for such a job at t is at most zero. Also, the total lag of Ψ is given by the sum of the lags of its jobs. Therefore, for $t > 0$, we have

$$\begin{aligned}
&\mathsf{LAG}(\Psi, t, \mathcal{S})\\
&= \sum_{\{\tau_{i,j} \text{ is in } \Psi\}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S})\\
&= \sum_{\substack{\{\tau_{i,j} \text{ is in } \Psi, \text{ and is}\\ \text{pending at } t^-\}}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) + \sum_{\substack{\{\tau_{i,j} \text{ is in } \Psi, \text{ and is}\\ \text{not pending at } t^-\}}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S})\\
&= \sum_{\substack{\{\tau_i \in \tau \mid \text{some}\\ \text{job of } \tau_i \text{ is in } \Psi\\ \text{and is pending at}\\ t^-\}}} \Bigg(\sum_{\substack{\{\tau_{i,j} \text{ is in } \Psi, \text{ and}\\ \text{is pending at } t^-\}}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) + \sum_{\substack{\{\tau_{i,j} \text{ is in } \Psi, \text{ and}\\ \text{is not pending at}\\ t^-\}}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S})\Bigg)\\
&\leq \sum_{\substack{\{\tau_i \in \tau \mid \text{some}\\ \text{job of } \tau_i \text{ is in } \Psi\\ \text{and is pending at}\\ t^-\}}} \Bigg(\sum_{\substack{\{\tau_{i,j} \text{ is in } \Psi, \text{ and}\\ \text{is pending at } t^-\}}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) + \sum_{\substack{\{\tau_{i,j} \text{ is in } \Psi, \text{ and}\\ \text{is not pending at}\\ t^-\}}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) + \sum_{\{\tau_{i,j} \text{ is not in } \Psi\}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S})\Bigg).
\end{aligned}$$

$$(4.13)$$

The last inequality holds because every job of τ_i that is not in Ψ is released after every job of the same task that is in Ψ. Hence, if some job of τ_i that is in Ψ is pending at t^-, then no later job can receive any allocation by t, and hence, the lag for every such job is at least zero. Because the lag of every task τ_i is given by the sum of the lags of its jobs, (4.13) implies the following.

$$\mathsf{LAG}(\Psi, t, \mathcal{S}) \leq \sum_{\substack{\{\tau_i \in \tau \mid \text{ some } \tau_{i,j} \text{ is in } \Psi, \\ \text{and is pending at } t^-\}}} \mathsf{lag}(\tau_i, t, \mathcal{S}) \tag{4.14}$$

Let the total *instantaneous utilization* of Ψ at time t be defined as the sum of the utilizations of all the tasks with an active job at t in Ψ:

$$U_{sum}(\Psi, t) = \sum_{\substack{\{\tau_i \in \tau \,:\, \tau_{i,j} \text{ is in } \Psi \\ \text{and is active at } t\}}} u_i \tag{4.15}$$

$$\Rightarrow \quad U_{sum}(\Psi, t) \leq U_{sum}(\tau). \tag{4.16}$$

The counterparts of (4.8) and (4.12) for job sets can then be expressed as follows. (As with (4.8) and (4.12), $t_1 < t_2$.)

$$\mathsf{A}(\mathsf{PS}_\tau, \Psi, t_1, t_2) = \int_{t_1}^{t_2} U_{sum}(\Psi, t)\mathrm{d}t \tag{4.17}$$

$$\leq (t_2 - t_1) \cdot U_{sum}(\tau) \tag{4.18}$$

$$\mathsf{LAG}(\Psi, t_2, \mathcal{S}) = \mathsf{LAG}(\Psi, t_1, \mathcal{S}) + \mathsf{A}(\mathsf{PS}_\tau, \Psi, t_1, t_2) - \mathsf{A}(\mathcal{S}, \Psi, t_1, t_2) \tag{4.19}$$

Definition 4.4 (busy and non-busy intervals): A time interval $[t, t+\delta)$, where $\delta > 0$, is said to be *busy* for Ψ if all M processors are executing some job in Ψ at each instant in the interval, *i.e.*, no processor is ever idle in the interval or executes a job not in Ψ. An interval $[t, t+\delta)$ that is not busy for Ψ is said to be *non-busy* for Ψ. $[t, t+\delta)$ is *continually non-busy* if every time instant in $[t, t+\delta)$ is non-busy, and is *maximally non-busy* if $[t, t+\delta)$ is continually non-busy and either $t = 0$ or t^- is busy.

If at least $U_{sum}(\Psi, t)$ jobs from Ψ are executing at every instant t in $[t_1, t_2)$ in a schedule $\mathcal{S}$, then, by (4.17), the total allocation in $\mathcal{S}$ to jobs in Ψ is at least the allocation that Ψ receives in a PS schedule. Therefore, by (4.19), the LAG of Ψ at t_2 cannot exceed that at t_1, and we have the following lemma.

Lemma 4.1 *If* $\mathsf{LAG}(\Psi, t+\delta, \mathcal{S}) > \mathsf{LAG}(\Psi, t, \mathcal{S})$*, where* $\delta > 0$ *and* $\mathcal{S}$ *is a schedule for* τ*, then* $[t, t+\delta)$ *is a non-busy interval for* Ψ*. Furthermore, there exists at least one instant* t' *in* $[t, t+\delta)$ *at which fewer than* $U_{sum}(\Psi, t')$ *tasks are executing jobs from* Ψ*.*

Definition 4.5 (continually-increasing LAG): If $\mathsf{LAG}(\Psi, t', \mathcal{S}) > \mathsf{LAG}(\Psi, t'^{-}, \mathcal{S})$ for all t' in $(t, t+\delta]$, then $[t, t+\delta)$ is said to be an interval with *continually-increasing* LAG for Ψ in $\mathcal{S}$.

If at least $U_{sum}(\Psi, t^-)$ jobs from Ψ are executing at t^- in $\mathcal{S}$, then by (4.17) and (4.19) again, $\mathsf{LAG}(\Psi, t, \mathcal{S}) \leq \mathsf{LAG}(\Psi, t^-, \mathcal{S})$, and the lemma below follows.

Lemma 4.2 *If* $[t, t+\delta)$*, where* $\delta > 0$*, is an interval across which* LAG *for* Ψ *is continually increasing, then fewer than* $U_{sum}(\Psi, t')$ *tasks execute jobs from* Ψ *at every instant* t'*, where* $t \leq t' < t + \delta$*.*

4.3.2 Deriving a Tardiness Bound

In this section, we show that on M processors, EDF-P-NP ensures a tardiness not exceeding $x + e_k$ for every task τ_k of every task system with total utilization at most M, where

$$x = \frac{\sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i - e_{\min}}{M - \sum_{i=1}^{\Lambda-\rho} \mu_i}, \tag{4.20}$$

and ρ is as defined in (4.21). Our proof is by contradiction. So, assume to the contrary that there exists some concrete task system with a job whose tardiness under EDF-P-NP exceeds x plus its worst-case execution cost. Let τ be one such concrete task system defined as follows.

Definition 4.6 (τ): τ is a concrete instantiation of a non-concrete task system τ^N, $\tau_{\ell,j}$ is a job of a task τ_ℓ in τ, $t_d = d_{\ell,j}$ ($\tau_{\ell,j}$'s deadline), and $\mathcal{S}$ is an EDF-P-NP schedule for τ such that (P1)–(P4) defined below hold.

- **(P1)** The tardiness of $\tau_{\ell,j}$ in $\mathcal{S}$ is greater than $x + e_\ell$.
- **(P2)** No concrete instantiation of τ^N satisfying (P1) releases fewer jobs than τ.
- **(P3)** No concrete instantiation of τ^N satisfying (P1) and (P2) has a job whose actual execution time is less than that of its corresponding job in τ.

(P4) The tardiness of every job of every task τ_k in τ with deadline less than t_d is at most $x + e_k$ in $\mathcal{S}$, where $x \geq 0$.

(P2) and (P3) can be thought of as identifying a minimal task system in the sense of releasing the minimum number of jobs and having the smallest actual job execution costs for which the claimed tardiness bound does not hold. It is easy to see that if the claimed bound does not hold for all task systems, then some task system satisfying (P2) and (P3) necessarily exists. (P4) identifies a job with the earliest deadline in τ for which the bound does not hold. In what follows, we show that if τ as defined above exists, then (4.20) is contradicted, thereby proving Theorem 4.1.

The completion time of $\tau_{\ell,j}$, and hence its tardiness, depends on the amount of work pending for $\tau_{\ell,j}$ at t_d, and the amount of work that can compete with $\tau_{\ell,j}$ after t_d. Hence, to meet our objective, we follow the steps below.

(S1) Determine an upper bound (UB) on the sum of the work pending for $\tau_{\ell,j}$ at t_d and the pending work due to other jobs that can compete with $\tau_{\ell,j}$ after t_d.

(S2) Determine a lower bound (LB) on the amount of such work required for the tardiness of $\tau_{\ell,j}$ to exceed $x + e_\ell$.

(S3) Show that unless (4.20) is contradicted, $\mathsf{UB} \leq \mathsf{LB}$, in turn showing that *tardiness*$(\tau_{\ell,j})$ is at most $x + e_\ell$, which is a contradiction to (P1).

To facilitate computing LB and UB, we define Ψ and $\overline{\Psi}$ as follows.

$\Psi \stackrel{\text{def}}{=}$ set of all jobs with deadlines at most t_d of tasks in τ

$\overline{\Psi} \stackrel{\text{def}}{=}$ set of all jobs of τ that are not in Ψ (*i.e.*, jobs with deadlines later than t_d)

Under $\mathsf{EDF\text{-}P\text{-}NP}$, the sum of the work pending for $\tau_{\ell,j}$ and competing work for $\tau_{\ell,j}$ at t_d is given by **(i)** the amount of work pending at t_d for jobs in Ψ plus **(ii)** the amount of work demanded after t_d by the non-preemptive sections of jobs that are not in Ψ but whose execution commenced before t_d. Because the deadline of every job in Ψ is at most t_d, all jobs in Ψ complete execution by t_d in $\mathsf{PS}\tau$. Hence, component (i) is given by $\mathsf{LAG}(\Psi, t_d, \mathcal{S})$. To facilitate computing component (ii), which will be denoted $\mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$, we define the following.

Definition 4.7 (priority inversions, blocking jobs, and blocked jobs): Under EDF-P-NP, a *priority inversion* occurs when a ready, higher-priority job waits while one or more lower-priority jobs execute in non-preemptive sections. Under such scenarios, the waiting higher-priority jobs are said to be *blocked* jobs, while the executing lower-priority jobs, *blocking* jobs. Note that a pending, higher-priority job is not considered blocked unless it is ready (*i.e.*, no prior job of the same task is pending).

Definition 4.8 (blocking and non-blocking, non-busy intervals): Recall that in a non-busy interval for Ψ, fewer than M jobs from Ψ execute. In an EDF-P-NP schedule, such a non-busy interval for Ψ can be classified into two types depending on whether a job in $\overline{\Psi}$ is executing while a ready job from Ψ is waiting. We will refer to the two types as *blocking* and *non-blocking* non-busy intervals. A *blocking, non-busy interval* is one in which a job in $\overline{\Psi}$ is executing while a ready job from Ψ is waiting, whereas a *non-blocking, non-busy interval* is one in which fewer than M jobs from Ψ are executing, but there does not exist a ready job in Ψ that is waiting. Definitions of maximal versions of these intervals are analogous to that of a maximally non-busy interval given in Definition 4.4.

Definition 4.9 (pending blocking jobs ($\mathcal{B}$) and work (B)): The set of all jobs in $\overline{\Psi}$ that commence executing a non-preemptive section before t and may continue to execute the same non-preemptive section at t in $\mathcal{S}$ is denoted $\mathcal{B}(\tau, \Psi, t, \mathcal{S})$ and the total amount of work pending at t for such non-preemptive sections is denoted $\mathsf{B}(\tau, \Psi, t, \mathcal{S})$.

Lastly, we define ρ as follows. Recall that apart from task system parameters, x in (4.20) also depends on ρ. The higher the value for ρ, the lower the tardiness. ρ is at least zero, and the current best-known value for ρ is given by Lemma 4.4.

Definition 4.10 (ρ):

$$\rho = \begin{cases} \text{Minimum number of jobs with deadlines at or after } t' \text{ guaranteed to execute at } t'^{-}, \text{ where } t < t' \leq t_d, \text{ and } [t, t') \text{ is a non-blocking, non-busy interval with continually-increasing } \mathsf{LAG} & , \text{if } b_{\max} = 0 \\ 0 & , \text{if } b_{\max} > 0 \end{cases} \tag{4.21}$$

With needed definitions in place, we now turn to deriving a tardiness bound following steps

(S1)–(S3). We first consider (S2), that of determining a lower bound on the sum of pending work and competing work required at t_d for tardiness of $\tau_{\ell,j}$ to exceed $x + e_\ell$. (We sometimes omit specifying the schedule $\mathcal{S}$ in functions lag, LAG, and B when unambiguous.)

4.3.2.1 Lower Bound on $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$ (Step (S2))

The required lower bound is established in the following lemma.

Lemma 4.3 *If* $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S}) \leq M \cdot x + e_\ell$*, then tardiness*$(\tau_{\ell,j}, \mathcal{S})$ *is at most* $x + e_\ell$*.*

Proof: Before proving the lemma, we establish the following claim.

> **Claim 4.1** *If* $\tau_{\ell,j}$ *executes at or after* t_d*, then it cannot be subsequently preempted.*
>
> **Proof:** Suppose the claim does not hold and let $t' > t_d$ denote the earliest time at which $\tau_{\ell,j}$ is preempted after executing at or after t_d. Note that only jobs with deadlines at or before t_d can preempt $\tau_{\ell,j}$. Since no job with eligibility time at t' can have its deadline at or before t_d, $\tau_{\ell,j}$ cannot be preempted at t' by a newly-arriving job. Our assumption that ties are resolved consistently implies that if $\tau_{\ell,j}$ is scheduled in preference to another job with an equal priority before t', then that job cannot preempt $\tau_{\ell,j}$ afterward. Hence, $\tau_{\ell,j}$ is preempted by a job with deadline at or before t_d that is not ready until t' because its predecessor executed until t'. (Otherwise, such a job can preempt $\tau_{\ell,j}$ before t'.) However, for every such job, the processor on which its predecessor executed is available for it at t'. Hence, no such job needs to preempt $\tau_{\ell,j}$. $\square$

With the above claim in place, to prove the lemma, we show that $\tau_{\ell,j}$ completes executing by $t_d + x + e_\ell$. If $j > 1$, then $d_{\ell,j-1} \leq t_d - p_\ell$ holds, and hence by (P4), we have the following.

(R) $\tau_{\ell,j-1}$ completes executing by $t_d - p_\ell + x + e_\ell$, for $j > 1$.

Let $\delta_\ell < e_\ell$ denote the amount of time that $\tau_{\ell,j}$ has executed for before time t_d. Then, the amount of work pending for $\tau_{\ell,j}$ at t_d is $e_\ell - \delta_\ell$. Recall that the total amount of work pending at t_d for jobs in Ψ and the non-preemptive sections of jobs in $\overline{\Psi}$ that commenced execution before t_d (*i.e.*, of jobs in $\mathcal{B}(\tau, \Psi, t_d, \mathcal{S})$) is given by $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$, which, by the statement of the lemma, is at most $M \cdot x + e_\ell$. Let $y \stackrel{\text{def}}{=} x + \delta_\ell / M$. Let $\mathcal{B}_J$ denote the set of all non-preemptive sections of jobs in $\mathcal{B}(\tau, \Psi, t_d, \mathcal{S})$ that commenced execution before t_d in $\mathcal{S}$

and are still pending at t_d. At the risk of abusing terms, let a time interval after t_d in which each processor is busy executing a job of Ψ or a non-preemptive section in $\mathcal{B}_J$ be referred to as a busy interval for Ψ and $\mathcal{B}_J$. We consider the following two cases.

Case 1: $[t_d, t_d + y)$ is a busy interval for Ψ and $\mathcal{B}_J$. In this case, the amount of work completed in $[t_d, t_d + y)$ for jobs in Ψ and non-preemptive sections in $\mathcal{B}_J$ is exactly $M \cdot y = M \cdot x + \delta_\ell$, and so, the amount of work pending at $t_d + y$ for jobs in Ψ and non-preemptive sections in $\mathcal{B}_J$ is at most $M \cdot x + e_\ell - (M \cdot x + \delta_\ell) = e_\ell - \delta_\ell$. Hence, if $\tau_{\ell,j}$ does not execute in $[t_d, t_d + y)$, then this pending work corresponds to that of $\tau_{\ell,j}$. That is, no job that can compete with $\tau_{i,j}$ is pending at $t_d + y$. Thus, the latest time that $\tau_{\ell,j}$ resumes (or begins, if $\delta_\ell = 0$) execution after t_d is at time $t_d + y$. By the claim above, $\tau_{\ell,j}$ cannot be preempted once it commences execution after t_d. Hence, $\tau_{\ell,j}$ completes execution at or before $t_d + y + e_\ell - \delta_\ell \leq t_d + x + e_\ell$.

Case 2: $[t_d, t_d + y)$ is not a busy interval for Ψ and $\mathcal{B}_J$. Let t' denote the first non-busy instant in $[t_d, t_d + y)$. This implies the following.

- **(J)** No job in $\overline{\Psi}$ begins executing a non-preemptive section or executes in a preemptive section in $[t_d, t')$; at most $M - 1$ tasks have non-preemptive sections that commenced before t_d or jobs in Ψ pending at or after t'.

Hence, no job of Ψ can be blocked by a job in $\overline{\Psi}$ at or after t'. Therefore, if $\tau_{\ell,j}$ has not completed executing before $t_d + y$, then either $\tau_{\ell,j}$ or a prior job of τ_ℓ should be executing at t'. If $\tau_{\ell,j}$ is executing at t', then because $t' < t_d + y$ holds, $\tau_{\ell,j}$ will complete executing before $t_d + y + e_\ell - \delta_\ell \leq t_d + x + e_\ell$. The remaining possibility is that $j > 1$ holds, and that a job of τ_ℓ that is prior to $\tau_{\ell,j}$ is executing at t'. In this case, $\tau_{\ell,j}$ could not have executed before t_d, and hence $\delta_\ell = 0$ and $y = x$ holds. Thus, $t' < t_d + y = t_d + x$ holds. Let t_c denote the time at which $\tau_{\ell,j-1}$ completes executing. Then, by (R), $t_c \leq t_d - p_\ell + x + e_\ell \leq t_d + x$ holds. Because some prior job of τ_ℓ is excuting at t', $t_c \geq t'$ holds. Hence, by (J), $\tau_{\ell,j}$ can commence execution at $t_c \leq t_d + x$ (on the same processor as that on which $\tau_{\ell,j-1}$ executed), and hence, can complete executing by $t_d + x + e_\ell$. ■

We next determine an upper bound on the sum of the pending work and the competing work described above, *i.e.*, $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$ (step (S1)).

4.3.2.2 Upper Bound on $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$

By Lemma 4.1, the LAG of Ψ can increase only across a non-busy interval for Ψ. Similarly, note that if $\mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$ is non-zero, then one or more jobs in $\overline{\Psi}$ should be executing in non-preemptive sections at t_d^-, that is, t_d^- should be a non-busy instant for Ψ. Hence, to determine an upper bound on the value that we are seeking, it suffices to consider only non-busy intervals for Ψ in $[0, t_d)$. As will be seen, an upper bound can be arrived at by reasoning about the number of tasks that can execute in and just before a non-busy interval and their lags. Towards this end, we prove a few lemmas below.

The lemma that follows identifies the existence of a class of jobs executing in a non-busy interval when no task has a non-preemptive segment. This lemma is somewhat technical in nature and is proved in an appendix. As noted earlier, this lemma provides a value to use for ρ, defined in (4.21), when $b_{\max} = 0$. In its absence, a weaker value of zero may be used.

Lemma 4.4 *Let $[t, t')$, where $t' > t$ and $t' \leq t_d$, be a non-blocking, non-busy interval across which* LAG *is continually increasing. If $b_{\max} = 0$, then there exists at least one job J such that J's deadline is at or after t' and J executes throughout $[\hat{t}, t')$, where $t \leq \hat{t} < t'$.*

Because the total system lag, LAG, for τ is given by the sum of the lags of its constituent tasks, towards determining an upper bound on LAG, we determine upper bounds on the lags of individual tasks. The lemma that follows shows that the lag of any task not executing continuously in a non-blocking, non-busy interval cannot exceed zero at the end of the interval.

Lemma 4.5 *Let $[t, t')$ be a continually non-blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$ and let τ_k be a task in τ with a job in Ψ that is active or pending at t'^-. If τ_k does not execute continuously in $[t, t')$* (i.e., *throughout $[t, t')$*), *then* $\mathsf{lag}(\tau_k, t') \leq 0$.

Proof: Because $[t, t')$ is continually non-busy and is non-blocking, at every instant in this interval, at least one processor is idle or a job in $\overline{\Psi}$ is executing while no job in Ψ is waiting. The absolute deadline of a job in $\overline{\Psi}$ is after t_d. Hence, if τ_k is not executing at t'^-, then it has no pending work at t'^-, and hence, its lag at t' is at most zero. On the other hand, suppose τ_k is executing at t'^-, but was not executing some time earlier. In this case, let t'', where $t < t'' < t'$, denote the latest time in (t, t') that τ_k transitions from a non-executing to an executing state. Because $[t, t')$ is continually non-busy and non-blocking, and τ_k is not executing at t''^-, jobs of τ_k with eligibility times, and hence, release times, before t'' complete execution in $\mathcal{S}$ before t'', and a job of τ_k is released (or becomes eligible before its release time) at t''. Hence, the total allocation to τ_k in $[0, t'')$ in $\mathcal{S}$ is at least that in PS_τ, and so, $\mathsf{lag}(\tau_k, t'', \mathcal{S}) \leq 0$. Since τ_k executes continuously in $[t'', t')$, we have $\mathsf{A}(\mathcal{S}, \tau_k, t'', t') = t' - t''$. Hence, $\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t'', \mathcal{S}) + \mathsf{A}(\mathsf{PS}_\tau, \tau_k, t'', t') - \mathsf{A}(\mathcal{S}, \tau_k, t'', t') \leq (t' - t'') \cdot u_k - (t' - t'') \leq 0$. ■

The next lemma bounds the lag of an arbitrary task at any arbitrary time at or before t_d.

Lemma 4.6 *Let t be an arbitrary time instant at or before t_d. Let τ_k be a task in τ and $\tau_{k,q}$ its earliest pending job at t. If $d_{k,q} \geq t$ holds, then $\mathsf{lag}(\tau_k, t, \mathcal{S}) \leq e_k$, else $\mathsf{lag}(\tau_k, t, \mathcal{S}) \leq x \cdot u_k + e_k$.*

Proof: Let $\delta_{k,q} < e_k$ denote the amount of time that $\tau_{k,q}$ executed for before t. We determine the lag of τ_k at t by considering two cases depending on $d_{k,q}$.

Case 1: $\boldsymbol{d_{k,q} < t}$. In PS_τ, $\tau_{k,q}$ completes execution at $d_{k,q}$, and the jobs of τ_k succeeding $\tau_{k,q}$ are allocated a share of u_k in every instant in their windows. Because different windows of the same task do not overlap, τ_k is allocated a share of u_k in every instant in $[d_{k,q}, t)$ in which it is active. Thus, the under-allocation to τ_k in $\mathcal{S}$ in $[0, t)$ is equal to the sum of the under-allocation to $\tau_{k,q}$ in $\mathcal{S}$, which is $e_k - \delta_{k,q}$, and the allocation, if any, to later jobs of τ_k in $[d_{k,q}, t)$ in PS_τ. Hence, we have

$$\mathsf{lag}(\tau_k, t, \mathcal{S}) \leq e_k - \delta_{k,q} + (t - d_{k,q}) \cdot u_k. \tag{4.22}$$

If $\tau_{k,q}$ executes for a full e_k time units, then the earliest time that it can complete execution is $t + e_k - \delta_{k,q}$. Because $t \leq t_d$, the deadline of $\tau_{k,q}$ is before t_d. Hence, by (P1), the tardiness of $\tau_{k,q}$ is at most $x + e_k$. Therefore, $t + e_k - \delta_{k,q} - d_{k,q} \leq x + e_k$, *i.e.*, $t - d_{k,q} \leq x + \delta_{k,q}$ holds. Substituting this value in (4.22), we have $\mathsf{lag}(\tau_k, t, \mathcal{S}) \leq x \cdot u_k + e_k + \delta_{k,q}(u_k - 1)$. The lemma follows because $u_k \leq 1$.

Case 2: $\boldsymbol{d_{k,q} \geq t}$. In this case, the amount of work done by PS_τ on $\tau_{k,q}$ up to time t is given by $e_k - (d_{k,q} - t) \cdot u_k$. Because all prior jobs of τ_k complete execution by t in both $\mathcal{S}$ and PS_τ, and $\tau_{k,q}$ has executed for $\delta_{k,q}$ time units before t, $\mathsf{lag}(\tau_k, t, \mathcal{S}) = e_k - (d_{k,q} - t) \cdot u_k - \delta_{k,q} \leq e_k - \delta_{k,q} \leq e_k$. ■

We next prove two claims, which will be used in proving later lemmas.

Claim 4.2 *Let $[t, t')$ be a maximally blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$. Then, the following hold.* **(i)** *$t > 0$;* **(ii)** *any job that is in $\overline{\Psi}$ and is executing at $\hat{t}$ in $[t, t')$ executes a single non-preemptive section continuously in $[t^-, \hat{t}]$.*

Proof: Since every job of Ψ has an earlier deadline than a job in $\overline{\Psi}$, a job in Ψ cannot be blocked at time 0 by a job in $\overline{\Psi}$. Therefore $t > 0$ holds. By the nature of $[t, t')$, no job of Ψ (including jobs that are blocked at t) is blocked by a job in $\overline{\Psi}$ at t^-. Hence, it cannot be the case that a job in Ψ is blocked at t due to a job in $\overline{\Psi}$ commencing execution of a non-preemptive section at t. Rather, the blocking non-preemptive section should have commenced execution before t and some blocked job becomes ready at t. Similarly, since every instant in $[t, t')$ is a blocking instant at which one or more ready jobs of Ψ are waiting, no job in $\overline{\Psi}$ can be executing in a preemptive section and no non-preemptive section of a job in $\overline{\Psi}$ can commence execution in (t, t'). The claim follows from these facts. ∎

Claim 4.3 *Let $[t, t')$ be a maximally blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$ such that $\mathsf{LAG}(\Psi, t') > \mathsf{LAG}(\Psi, t)$. Then, at t and t^-, at most Λ tasks have jobs in Ψ that are executing. (Note that the tasks executing at t and t^- need not be the same.)*

Proof: Because $\mathsf{LAG}(\Psi, t') > \mathsf{LAG}(\Psi, t)$ holds, by Lemma 4.1, (4.16), and (4.6), there exists at least one time instant $\hat{t}$ in $[t, t')$ such that at most Λ tasks have executing jobs in Ψ at $\hat{t}$. Let $k \leq \Lambda$ denote the number of such tasks. Then, since $[t, t')$ is a blocking, non-busy interval, no processor is idle in the interval. Hence, exactly $M - k$ jobs from $\overline{\Psi}$ are executing at $\hat{t}$. By Claim 4.2, $t > 0$ and each of the $M - k$ jobs is executing continuously in $[t^-, \hat{t}]$. Hence, at t^- and t, at most $k \leq \Lambda$ tasks may have executing jobs in Ψ. ∎

In completing the derivation, we make use of the following two lemmas, which are proved in Appendix A. These lemmas are purely mathematical and involve manipulations of task execution and non-preemptive segment costs and are independent of scheduling rules.

Lemma 4.7 *The following properties hold for sets $\Gamma(k + c, \ell)$ and $\Pi(k + c, \ell)$ with $0 \leq \ell \leq k \leq N$ and $0 \leq c \leq N - k$, where Γ and Π are as defined in (4.3) and (4.4), respectively.*

(i) $\sum_{\tau_i \in \Gamma(k+c,\ell)} e_i + \sum_{\tau_i \in \Pi(k+c,\ell)} b_i \leq \sum_{\tau_i \in \Gamma(k,\ell)} e_i + \sum_{\tau_i \in \pi(k,\ell)} b_i + \sum_{i=1}^{c} \beta_i$.

(ii) $\sum_{\tau_i \in \Gamma(k+c,\ell)} e_i + \sum_{\tau_i \in \Pi(k+c,\ell)} b_i \geq \sum_{\tau_i \in \Gamma(k,\ell)} e_i + \sum_{\tau_i \in \Pi(k,\ell)} b_i$.

Lemma 4.8 *Let α and β be any two **disjoint** subsets of tasks in τ such that $|\alpha| \leq \ell$ and $|\alpha| + |\beta| \leq k$, where $0 \leq \ell \leq k \leq N$. Then, $\sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta} b_i \leq \sum_{\tau_i \in \Gamma(k,\ell)} e_i + \sum_{\tau_i \in \Pi(k,\ell)} b_i$.*

The next two lemmas show how to bound LAG at the end of a non-blocking, non-busy interval, and $\mathsf{LAG} + \mathsf{B}$ at the end of a blocking, non-busy interval.

Lemma 4.9 *Let $[t, t')$ be a non-blocking, non-busy interval in $[0, t_d)$ across which* LAG *is continually increasing. Let k denote the number of tasks that are executing jobs in Ψ at t'^{-}. Then, we have the following:* **(i)** $k \leq \Lambda$*; and* **(ii)** $\mathsf{LAG}(\Psi, t', \mathcal{S}) \leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda)}} e_i \leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i$.

Proof: Let α denote the subset of all tasks in τ that are executing jobs in Ψ continuously in $[t, t')$. Let γ be the subset of all tasks not in α, but which have jobs executing at t'^{-}. Then, by the statement of the lemma, $|\alpha| + |\gamma| = k$ holds. Because LAG is continually increasing across the interval $[t, t')$, by Lemma 4.2 and (4.16), we have

$$|\alpha| + |\gamma| = k < U_{sum}(\Psi, t') \leq U_{sum}(\tau). \tag{4.23}$$

Because k is an integer, by (4.23) and (4.6), we have $k \leq \Lambda$, which establishes Part (i).

Let Υ denote the set of all tasks that have a job with a deadline at or after t' that is executing at t'^{-}. Because $[t, t')$ is a non-blocking, non-busy interval across which LAG is continually increasing, by Definition 4.10,

$$|\Upsilon| \geq \rho. \tag{4.24}$$

Let Υ_α denote the set of all tasks in Υ that are in α. Then, tasks in $\Upsilon \setminus \Upsilon_\alpha$ (referred to as Υ_γ) are in γ (because they are executing at t'^{-}). That is, we have

$$\Upsilon_\alpha \subseteq \alpha, \tag{4.25}$$

$$\Upsilon \setminus \Upsilon_\alpha \subseteq \gamma. \tag{4.26}$$

Let

$$|\Upsilon_\alpha| = \rho_\alpha. \tag{4.27}$$

Then, we have the following.

$$|\Upsilon \setminus \Upsilon_\alpha| \geq \rho - \rho_\alpha \quad \text{(by (4.24) and (4.27))} \tag{4.28}$$

$$|\gamma| \geq \rho - \rho_\alpha \quad \text{(by (4.26) and (4.28))} \tag{4.29}$$

By (4.14), the LAG of Ψ at t' is at most the sum of the lags at t' of all tasks in τ with at least one job in Ψ that is pending at t'^{-}. By Lemma 4.5, the lag of such a task that does not execute continuously in $[t, t')$ is at most zero. Hence, to determine an upper bound on LAG of Ψ at t', it is sufficient to determine an upper bound on the sum of the lags of such tasks that are executing continuously in $[t, t')$, *i.e.*, tasks in α. Thus,

$$\begin{aligned}
\mathsf{LAG}(\Psi, t') &\leq \sum_{\tau_i \in \alpha} \mathsf{lag}(\tau_i, t', \mathcal{S}) \\
&= \sum_{\tau_i \in \Upsilon_\alpha} \mathsf{lag}(\tau_i, t', \mathcal{S}) + \sum_{\tau_i \in \alpha \setminus \Upsilon_\alpha} \mathsf{lag}(\tau_i, t', \mathcal{S}) \qquad \text{(by (4.25))}.
\end{aligned}$$

The definition of Υ implies that the deadline of the earliest pending job at t' of every task in that set is at or after t'. Therefore, by Lemma 4.6, the lag of every task in Υ is at most its execution cost. Hence,

$$\begin{aligned}
\mathsf{LAG}(\Psi, t') &\leq \sum_{\tau_i \in \Upsilon_\alpha} e_i + \sum_{\tau_i \in \alpha \setminus \Upsilon_\alpha} (x \cdot u_i + e_i) \qquad \text{(by the definition of } \Upsilon \text{ and Lemma 4.6)} \\
&= \sum_{\tau_i \in \alpha \setminus \Upsilon_\alpha} x \cdot u_i + \sum_{\tau_i \in \alpha} e_i \\
&\leq \sum_{i=1}^{k-\rho} x \cdot \mu_i + \sum_{\tau_i \in \alpha} e_i \qquad \text{(by (4.23), (4.29), (4.27), and the definition of } \mu_i) \\
&\leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \alpha} e_i \qquad (k \leq \Lambda, \text{ by Part (i)}) \\
&\leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(k)}} e_i \qquad \text{(by the definition of } \Gamma \text{ in (4.2))} \\
&\leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda)}} e_i \qquad (k \leq \Lambda, \text{ by Part (i)}) \qquad (4.30) \\
&= \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda,\Lambda)}} e_i \qquad \text{(by (4.2) and (4.3))} \\
&\leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} b_i \qquad \text{(by Lemma 4.7(ii))}. \qquad (4.31)
\end{aligned}$$

Part (ii) follows from (4.30) and (4.31). ■

Lemma 4.10 *Let $[t, t')$ be a maximally blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$ such that $k \leq \Lambda$ tasks have jobs in Ψ executing at t. Then, we have the following:* **(i)** $\mathsf{LAG}(\Psi, t') \leq$

$\sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma(\Lambda+1,\Lambda)} e_i + \sum_{\tau_i \in \Pi(\Lambda+1,\Lambda)} b_i$, *and* **(ii)** $\mathsf{LAG}(\Psi, t') + \mathsf{B}(\tau, \Psi, t') < \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma(M,\Lambda)} e_i + \sum_{\tau_i \in \Pi(M,\Lambda)} b_i$, *where* ρ *is as defined in (4.21).*

Proof: Let $\mathcal{J}$ denote the set of all jobs of $\overline{\Psi}$ that are executing in some non-preemptive section at t, and, hence, are blocking one or more jobs of Ψ. Let β denote the subset of all tasks in τ with jobs in $\mathcal{J}$. Since k jobs from Ψ are executing at t (by the statement of the lemma) and $[t, t')$ is a blocking, non-busy interval (hence, no processor is idle), we have

$$
\begin{aligned}
|\mathcal{J}| = |\beta| &= M - k && (4.32)\\
&\geq \Lambda + 1 - k \qquad \text{(by } U_{sum} \leq M \text{ and (4.6))}. && (4.33)
\end{aligned}
$$

By the definition of $[t, t')$ in the statement of the lemma and Claim 4.2, it follows that t^- is a non-blocking, non-busy instant. By (4.32) and Claim 4.2 again, it follows that at least $M - k$ jobs of $\overline{\Psi}$, including all the jobs in $\mathcal{J}$, are executing at t^-. Therefore, at most k jobs from Ψ can be executing at t^-. Let α denote the set of all tasks with jobs in Ψ that are executing at t^-. Hence,

$$|\alpha| \leq k. \qquad (4.34)$$

Note that by their definitions, sets α and β are disjoint and the following hold.

$$
\begin{aligned}
\alpha \cap \beta &= \emptyset && (4.35)\\
|\alpha| + |\beta| &\leq M && (4.36)
\end{aligned}
$$

Since no job of Ψ that is not executing at a non-blocking, non-busy instant can be pending (and hence no such job or its task may have a positive lag at t), by (4.14), we have the following.

$$
\begin{aligned}
\mathsf{LAG}(\Psi, t) &\leq \sum_{\substack{\{\tau_i \in \tau | \tau_{i,j} \text{ is in } \Psi, \text{ and} \\ \text{is executing at } t^-\}}} \mathsf{lag}(\tau_i, t, \mathcal{S})\\
&\leq \sum_{\substack{\{\tau_i \in \tau | \tau_{i,j} \text{ is in } \Psi, \text{ and} \\ \text{is executing at } t^-\}}} (x \cdot u_i + e_i) && \text{(by Lemma 4.6)}\\
&\leq \sum_{\tau_i \in \alpha} (x \cdot u_i + e_i) && \text{(by the definition of } \alpha\text{)} \qquad (4.37)
\end{aligned}
$$

By Claim 4.2, every job of $\overline{\Psi}$ that is executing anywhere in $[t, t')$ is in $\mathcal{J}$. Furthermore, every such job executes a single non-preemptive section in $[t^-, t')$. Hence, for any time u in

$[t, t')$, $\mathsf{B}(\tau, \Psi, u)$ is given by the amount of work pending at u for the non-preemptive sections executing at t of jobs in $\mathcal{J}$. Let τ_i be a task with a job J in $\mathcal{J}$. Then, the amount of work that can be pending for the non-preemptive section of J executing at t is less than b_i. Therefore, by the definition of set β, we have the following.

$$\mathsf{B}(\tau, \Psi, t) < \sum_{\tau_i \in \beta} b_i \tag{4.38}$$

Hence, by (4.37) and (4.38), we have

$$\begin{aligned}
&\mathsf{LAG}(\Psi, t) + \mathsf{B}(\tau, \Psi, t) \\
&\quad < \sum_{\tau_i \in \alpha} (x \cdot u_i + e_i) + \sum_{\tau_i \in \beta} b_i \\
&\quad = \sum_{\tau_i \in \alpha} x \cdot u_i + \sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta} b_i \\
&\quad \le \sum_{i=1}^{\Lambda} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(M,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(M,\Lambda)}} b_i && \text{(by Lemma 4.8, because by (4.32) – (4.36), } |\alpha| \le k \le \Lambda,\ |\alpha| + |\beta| \le M - k + |\alpha| \le M\text{)} \\
&\quad = \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(M,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(M,\Lambda)}} b_i && \text{(}[t,t')\text{ is a blocking interval, so } b_{\max} > 0\text{, and by (4.21), } \rho = 0\text{).}
\end{aligned} \tag{4.39}$$

Finally, we are left with determining an upper bound on LAG and the sum of LAG and B at t'. We will determine an upper bound on LAG first. Let $\hat{t}$ denote the earliest time, if any, in (t, t') at which at least $\Lambda + 1$ jobs from Ψ are executing. If no such time exists, then let $\hat{t} = t'$. By the nature of $[t, t')$ and Claim 4.2, no job in $\overline{\Psi}$ can begin executing (either a preemptive or non-preemptive section) anywhere in $[t, t')$, and hence, at least $\Lambda+1$ jobs from Ψ are executing at each instant in $[\hat{t}, t')$. By (4.6) and (4.16), $\Lambda + 1 \ge U_{sum}(\tau) \ge U_{sum}(\Psi, u)$, for all u. Hence, by Lemma 4.1, LAG for Ψ does not increase across $[\hat{t}, t')$. That is,

$$\mathsf{LAG}(\Psi, t', \mathcal{S}) \le \mathsf{LAG}(\Psi, \hat{t}, \mathcal{S}). \tag{4.40}$$

Thus, to determine an upper bound on LAG at t^-, it suffices to determine an upper bound on LAG at $\hat{t}$.

Let X denote the total execution within $[t, t')$ of the first $\Lambda + 1 - k$ jobs in $\mathcal{J}$ to complete executing their non-preemptive sections (the actual completion time of these non-preemptive sections is immaterial). Note that X is well defined because by (4.33), $|\mathcal{J}| \ge \Lambda + 1 - k$ holds.

Because $[t, t')$ is maximally blocking, no processor is idle in $[t, \hat{t})$, and because k jobs in Ψ are executing at t, at least k jobs execute at every instant in $[t, t')$. Also, when a job in $\mathcal{J}$ completes executing its non-preemptive section in $[t, t')$, the processor it executed on is allocated to a waiting job in Ψ. Hence, the total time allocated to jobs in Ψ in $[t, \hat{t})$, $\mathsf{A}(\mathcal{S}, \Psi, t, \hat{t})$, is equal to $k \cdot (\hat{t} - t) + (\Lambda + 1 - k) \cdot (\hat{t} - t) - X = (\Lambda + 1) \cdot (\hat{t} - t) - X$. In PS_τ, jobs in Ψ could execute for at most $U_{sum}(\tau) \cdot (\hat{t} - t)$ time, *i.e.*, $\mathsf{A}(\mathsf{PS}_\tau, \Psi, t, \hat{t}) \le U_{sum}(\tau) \cdot (\hat{t} - t) \le (\Lambda + 1) \cdot (\hat{t} - t)$. (The second inequality is by (4.6).) Therefore, by (4.12),

$$\begin{aligned}
\mathsf{LAG}(\Psi, \hat{t}) &= \mathsf{LAG}(\Psi, t) + \mathsf{A}(\mathsf{PS}_\tau, \Psi, t, \hat{t}) - \mathsf{A}(\mathcal{S}, \Psi, t, \hat{t}) \\
&\le \mathsf{LAG}(\Psi, t) + (\Lambda + 1) \cdot (\hat{t} - t) - (\Lambda + 1) \cdot (\hat{t} - t) + X \\
&= \mathsf{LAG}(\Psi, t) + X. \qquad (4.41)
\end{aligned}$$

Let $\beta^{(\Lambda+1-k)}$ denote the subset of $\Lambda + 1 - k$ tasks whose jobs in $\mathcal{J}$ complete executing their non-preemptive sections the earliest. (Again, by (4.33), $|\mathcal{J}| \ge \Lambda + 1 - k$.) Then, by the discussion thus far,

$$\begin{aligned}
&\mathsf{LAG}(\Psi, t', \mathcal{S}) \\
&\quad \le \mathsf{LAG}(\Psi, \hat{t}, \mathcal{S}) && \text{(by (4.40))} \\
&\quad \le \mathsf{LAG}(\Psi, t) + X && \text{(by (4.41))} \\
&\quad \le \sum_{\tau_i \in \alpha} (x \cdot u_i + e_i) + \sum_{\tau_i \in \beta^{(\Lambda+1-k)}} b_i && \text{(by (4.37) and the definition of } X) \\
&\quad \le \sum_{i=1}^{k} x \cdot \mu_i + \sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta^{(\Lambda+1-k)}} b_i && \text{(by (4.34) and the definition of } \mu_i) \\
&\quad \le \sum_{i=1}^{\Lambda} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i && \begin{array}{l}\text{(by Lemma 4.8; note that } \alpha \text{ and } \beta^{(\Lambda+1-k)} \\ \text{are disjoint, } |\alpha| \le k \le \Lambda \text{ holds by (4.34),} \\ \text{and because } |\beta^{(\Lambda+1-k)}| = \Lambda + 1 - k,\ |\alpha| + \\ |\beta^{(\Lambda+1-k)}| \le \Lambda + 1 \text{ holds as well)}\end{array} \\
&\quad \le \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i && \text{(by (4.21), } \rho = 0 \text{ since } b_{\max} > 0).
\end{aligned}$$

The above establishes Part (i) of the lemma.

To determine an upper bound on LAG plus B at t', let $X' \le \mathsf{B}(\tau, \Psi, t)$ denote the total amount of time that jobs in $\mathcal{J}$ execute on all M processors in $[t, t')$. Then, the total time allocated to jobs in Ψ in $[t, t')$, $\mathsf{A}(\mathcal{S}, \Psi, t, t')$, is equal to $M \cdot (t' - t) - X'$. In PS_τ, jobs in

Ψ could execute for at most $U_{sum}(\tau) \cdot (t' - t)$ time, *i.e.*, $\mathsf{A}(\mathsf{PS}_\tau, \Psi, t, t') \le U_{sum}(\tau) \cdot (t' - t)$. Therefore, $\mathsf{LAG}(\Psi, t') = \mathsf{LAG}(\Psi, t) + \mathsf{A}(\mathsf{PS}_\tau, \Psi, t, t') - \mathsf{A}(\mathcal{S}, \Psi, t, t') \le \mathsf{LAG}(\Psi, t) + (U_{sum}(\tau) - M) \cdot (t' - t) + X' \le \mathsf{LAG}(\Psi, t) + X'$. Since jobs in $\mathcal{J}$ execute for a total time of X' in $[t, t')$, the pending work for non-preemptive sections of jobs in $\mathcal{J}$, and hence, those in $\mathcal{B}(\tau, \Psi, t')$ at t', *i.e.*, $\mathsf{B}(\tau, \Psi, t')$, is at most $\mathsf{B}(\tau, \Psi, t) - X'$. Thus, $\mathsf{LAG}(\Psi, t') + \mathsf{B}(\tau, \Psi, t') \le \mathsf{LAG}(\Psi, t) + \mathsf{B}(\tau, \Psi, t)$, which by (4.39), establishes Part (ii) of the lemma. ■

Finally, Lemmas 4.9 and 4.10 can be used to establish the following.

Lemma 4.11 *If t_d^- is **not** a non-blocking, non-busy instant, then*

$$\mathsf{LAG}(\Psi, t_d) + \mathsf{B}(\tau, \Psi, t_d) \le \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i,$$

where ρ is as defined in (4.21).

Proof: We consider the following two cases based on the nature of t_d^-.

Case 1: t_d^- is a busy instant. In this case, because t_d^- is busy, by definition, $\mathsf{B}(\tau, \Psi, t_d) = 0$. By Lemma 4.1, the LAG of Ψ can increase only across a non-busy interval. Therefore, LAG at t_d is at most that at the end of the latest non-busy instant before t_d. If no non-busy interval exists in $[0, t_d)$, then $\mathsf{LAG}(\Psi, t_d) \le \mathsf{LAG}(\Psi, 0) = 0$. Otherwise, if the latest non-busy interval before t_d is non-blocking, then by Part (ii) of Lemma 4.9, LAG at t_d is at most $\sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i$, establishing the lemma in this case. The remaining case is that the latest non-busy interval before t_d is blocking, in which case by Part (i) of Lemma 4.10, LAG is at most $\sum_{i=1}^{\Lambda-\rho}(x \cdot \mu_i) + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i$. The lemma thus holds in this case too.

Case 2: t_d^- is a blocking, non-busy instant. Let $t < t_d$ be the earliest instant before t_d such that $[t, t_d)$ is a maximally blocking, non-busy interval. Let k denote the number of tasks whose jobs in Ψ are executing at t. We consider the following two subcases.

Subcase 2(a): $\mathsf{LAG}(\Psi, t_d) > \mathsf{LAG}(\Psi, t)$ or $k \leq \Lambda$. If $\mathsf{LAG}(\Psi, t_d) > \mathsf{LAG}(\Psi, t)$ holds, then by Claim 4.3, $k \leq \Lambda$ holds. Hence, in either case, by Part (ii) of Lemma 4.10, we have

$$\begin{aligned}\mathsf{LAG}(\Psi, t_d) + \mathsf{B}(\tau, \Psi, t_d) &< \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(M,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(M,\Lambda)}} b_i \\ &\leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i,\end{aligned}$$

establishing the lemma for this subcase. The second inequality follows from Lemma 4.7(i) (because $M \geq \Lambda + 1$, by (4.6) and $M \leq U_{sum}$).

Subcase 2(b): $k > \Lambda$ and $\mathsf{LAG}(\Psi, t_d) \leq \mathsf{LAG}(\Psi, t)$. By the conditions of this subcase, it follows that LAG at t_d is bounded from above by the LAG at the end of the latest non-blocking or blocking non-busy interval before t across which LAG increases. If no such interval exists, then by Lemma 4.1, $\mathsf{LAG}(\tau, t_d) \leq \mathsf{LAG}(\tau, 0) = 0$. Otherwise, (as with Case 1) by Part (ii) of Lemma 4.9 and Part (i) of Lemma 4.10, we have

$$\mathsf{LAG}(\Psi, t_d) \leq \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i. \tag{4.42}$$

Since $k > \Lambda$ holds, at most $M - \Lambda - 1$ jobs from $\overline{\Psi}$ can be executing at t, and by Claim 4.2, at most $M - \Lambda - 1$ such jobs can be executing at t_d^- as well. The amount of time for which such a job of task τ_i can execute past t_d in its non-preemptive section is less than b_i. Thus, $\mathsf{B}(\tau, \Psi, t_d) < \sum_{i=1}^{M-\Lambda-1} \beta_i$ holds. Therefore, by (4.42), $\mathsf{LAG}(\Psi, t_d) + \mathsf{B}(\tau, \Psi, t_d) < \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i$ holds. ■

Lemma 4.12 *If t_d^- is a non-blocking, non-busy instant, then tardiness$(\tau_{\ell,j}) \leq x + e_\ell$.*

Proof: If t_d^- is a non-blocking, non-busy instant, then at most $M - 1$ tasks have pending jobs at t_d^- in Ψ and each of these tasks is executing at t_d^-. Let α denote the set of all such tasks. Since no job with a deadline at or before t_d can be released at or after t_d, a task in α cannot be preempted until all its jobs in Ψ complete execution, *i.e.*, each task in α executes continuously until all its jobs in Ψ complete execution. If $\tau_{\ell,j}$ completes execution before t_d, then its tardiness is zero. Otherwise, τ_ℓ is in α, and by Lemma 4.6, its lag at t_d is at most $x \cdot u_\ell + e_\ell$. Since the deadline of $\tau_{\ell,j}$ is at t_d, no job of τ_ℓ with a deadline after t_d is released before t_d. Jobs with release times after t_d receive no allocation before t_d in $\mathsf{PS}\tau$, even if their eligibility times are earlier. Hence, lag for such jobs of τ_ℓ is zero at t_d. Thus, τ_ℓ's lag at t_d (given by the sum of the lags of all its jobs) is at most the amount of work pending for its jobs in Ψ. Because τ_ℓ is in α, it executes continuously until all its jobs in Ψ, which includes $\tau_{\ell,j}$, complete execution. The latest completion time is given by $t_d+\mathsf{lag}(\tau_\ell, t_d, \mathcal{S}) \leq t_d+x\cdot u_\ell+e_\ell \leq t_d+x+e_\ell$, and hence, the tardiness of $\tau_{\ell,j}$ is at most $x + e_\ell$. ■

4.3.2.3 Finishing Up (Step (S3))

If t_d is a non-blocking, non-busy instant, then Lemma 4.12 contradicts (P1). Otherwise, by Lemma 4.3 and (P1), $\mathsf{LAG}(\tau, t_d, \mathcal{S}) + \mathsf{B}(\Psi, \tau, t_d, \mathcal{S}) > M \cdot x + e_\ell$. Therefore, by Lemma 4.11, $M \cdot x + e_\ell < \sum_{i=1}^{\Lambda-\rho} x \cdot \mu_i + \sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i$, *i.e.*,

$$\begin{aligned} x &< \frac{\sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i - e_\ell}{M - \sum_{i=1}^{\Lambda-\rho} \mu_i} \\ &\leq \frac{\sum_{\tau_i \in \Gamma^{(\Lambda+1,\Lambda)}} e_i + \sum_{\tau_i \in \Pi^{(\Lambda+1,\Lambda)}} b_i + \sum_{i=1}^{M-\Lambda-1} \beta_i - e_{\min}}{M - \sum_{i=1}^{\Lambda-\rho} \mu_i}, \end{aligned}$$

which contradicts (4.20). Thus, Theorem 4.1 below follows.

Theorem 4.1 *Let τ be a sporadic task system with non-preemptable segments and $U_{sum} \leq M$. Then, on M processors,* EDF-P-NP *ensures a tardiness of at most $x + e_k$ to every task τ_k in τ, where x is as defined by (4.20).*

If no task has non-preemptable segments, then EDF-P-NP reduces to g-EDF (fully-preemptive global EDF). In such a case, $b_i = \beta_i = 0$, for all i. Hence, $b_{\max} = 0$, and by Lemma 4.4, $\rho \geq 1$. Further, $e_i - b_i = e_i$, and hence, $\Gamma^{(\Lambda+1,\Lambda)}$ is simply a subset of Λ tasks of τ with the highest execution costs, *i.e.*, $\Gamma^{(\Lambda+1,\Lambda)} = \Gamma^{(\Lambda)}$. Therefore, a tardiness bound under g-EDF is given by the following corollary to Theorem 4.1.

Corollary 4.1 g-EDF *ensures a tardiness bound of*

$$\frac{\sum_{i=1}^{\Lambda} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda-1} \mu_i} + e_k \tag{4.43}$$

to every task τ_k of a sporadic task system τ with $U_{sum} \leq M$.

A sufficient condition on the individual task utilizations for a given tardiness bound under g-EDF that places no restriction on the total utilization can be derived from the above corollary, as given below.

Corollary 4.2 g-EDF *ensures a tardiness of at most* $x_p + e_k$ *for every task* τ_k *of a sporadic task system* τ *on* M *processors if the sum of the utilizations of the* $\Lambda - 1$ *tasks with highest utilizations,* $\sum_{i=1}^{\Lambda-1} \mu_i$*, is at most* $M - \frac{\sum_{i=1}^{\Lambda} \epsilon_i + e_{\min}}{x_p}$ *and* $U_{sum}(\tau) \leq M$*.*

Proof: By Corollary 4.1, the tardiness for task τ_k of τ in a g-EDF schedule is at most $\frac{\sum_{i=1}^{\Lambda} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda-1} \mu_i} + e_k$. Therefore, a tardiness not exceeding $x_p + e_k$ can be guaranteed if $\frac{(\sum_{i=1}^{\Lambda} \epsilon_i) - e_{\min}}{M - \sum_{i=1}^{\Lambda-1} \mu_i} + e_k \leq x_p + e_k$ holds. On rearranging the terms, we arrive at the condition in the corollary. ■

Similarly, EDF-P-NP reduces to g-NP-EDF (fully-non-preemptive EDF) if $b_i = e_i$, for all i. Therefore, $\Gamma^{(\Lambda+1,\Lambda)} \cup \Pi^{(\Lambda+1,\Lambda)} = \Gamma^{(\Lambda+1)}$. Further, by (4.21), $\rho = 0$ holds. In this case, the tardiness bound of Theorem 4.1 reduces to the following.

Corollary 4.3 *On* M *processors,* g-NP-EDF *ensures a tardiness bound of*

$$\frac{\sum_{i=1}^{\Lambda+1} \epsilon_i + \sum_{i=1}^{M-\Lambda-1} \beta_i - e_{\min}}{M - \sum_{i=1}^{\Lambda} \mu_i} + e_k$$

to every task τ_k *of a sporadic task system* τ *with* $U_{sum} \leq M$*.*

4.3.3 Tardiness Bound under g-EDF for Two-Processor Systems

If $1 < U_{sum} \leq 2$, then the tardiness bound under g-EDF given by Corollary 4.1 for task τ_k under two processors (*i.e.*, when $M = 2$) is $(e_{\max} - e_{\min})/2 + e_k$. However, an improved bound of $e_{\max}/2 + e_k/2$ can be ensured if we note that $\mathsf{LAG} + \mathsf{B}$ at t_d, as given by Lemma 4.11, is at most $e_{\max}$, which is independent of x. This improved bound is derived in the following theorem.

Theorem 4.2 *On two processors,* g-EDF *ensures a tardiness of at most* $\frac{e_{\max} + e_k}{2}$ *for every task* τ_k *of every sporadic task system with* $U_{sum} \leq 2$*.*

Proof: Suppose that the theorem does not hold. Then, there exists a concrete instantiation τ of a non-concrete task system with $U_{sum} \leq 2$ such that $\tau_{\ell,j}$ is a job of task τ_ℓ in τ, $\mathcal{S}$ is a g-EDF schedule for τ, $t_d = d_{\ell,j}$, and (P1)–(P4) (defined in the beginning of Section 4.3.2) hold, where $x = \frac{e_{\max} - e_\ell}{2}$. As with the general case, if t_d^- is a non-blocking, non-busy instant, then

Lemma 4.12 contradicts (P1). Otherwise, by Lemmas 4.11 and 4.4, and because $b_i = 0$, for all i, we have

$$\mathsf{LAG}(\tau, t_d, \mathcal{S}) + \mathsf{B}(\Psi, \tau, t_d, \mathcal{S}) \leq e_{\max}. \tag{4.44}$$

By (P1), *tardiness*$(\tau_{\ell,j}) > x + e_\ell$. Hence, by the contrapositive of Lemma 4.3, we have $\mathsf{LAG}(\tau, t_d, \mathcal{S}) + \mathsf{B}(\Psi, \tau, t_d, \mathcal{S}) > 2 \cdot x + e_\ell$, which, by (4.44), implies that $e_{\max} > 2 \cdot x + e_\ell = e_{\max}$, a contradiction. The theorem, therefore, follows. ■

4.3.4 Improving Accuracy and Speed

In this subsection, we discuss possible improvements to the accuracy of the tardiness bounds of g-EDF and g-NP-EDF given in Corollaries 4.1 and 4.3, respectively, and the time required to compute them. We will refer to the bounds given by Corollaries 4.1 and 4.3 as EDF-BASIC and NP-EDF-BASIC, respectively.

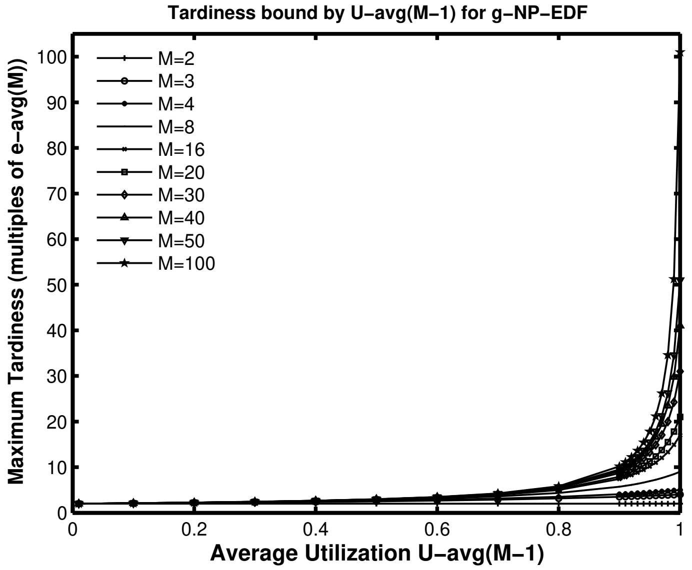
Improving accuracy. If the time taken to compute a tardiness bound, or sufficient task utilizations for a given tardiness bound, is not a concern, then some improvement to the values may be possible by relaxing a pessimistic assumption that we make in Lemmas 4.9 and 4.10. We not only assume that the tasks that are executing at the end of the non-blocking, non-busy interval in Lemma 4.9, or just before the beginning of the blocking, non-busy interval in Lemma 4.10, have the highest utilizations, but also that they have the highest execution costs. For g-EDF, this can be relaxed by sorting tasks by non-increasing order of $x \cdot u_k + e_k$ (where, as defined in (P4), the tardiness of τ_k is at most $x + e_k$) and by using the utilizations and execution costs of the top $\Lambda - 1$ tasks in the expression in Corollary 4.1. (The Λ^{th} execution cost should be taken as the maximum of the execution costs of the remaining tasks.) If x is known (*i.e.*, when determining utilization restrictions for a given tardiness), as in applying Corollary 4.2, then this procedure is straightforward. Nevertheless, even when seeking x (*i.e.*, when determining a tardiness bound), as in Corollary 4.1, an iterative procedure could be used. In this iterative procedure, the bound given by EDF-BASIC will be used as the initial value for x. This initial value will then be used to determine the set of tasks whose utilizations and execution costs should be used in improving x. The procedure should be continued until the task set converges. (It is easy to show that convergence is guaranteed.) We will refer to the bound computed using such an iterative procedure as EDF-ITER. This procedure is illustrated in the example below.

Example 4.2. Let τ be a task system consisting of the following eight tasks with $U_{sum} = 4.0$ scheduled under g-EDF on $M = 4$ processors: $\tau_1(15, 150)$–$\tau_4(15, 150)$, $\tau_5(9, 10)$–$\tau_8(9, 10)$. For this task set, $\mu_1 = \mu_2 = 0.9$, $\epsilon_1 = \epsilon_2 = \epsilon_3 = 15$, and $\Lambda = 3$. Therefore, a tardiness bound for τ_k obtained using EDF-BASIC is $(45 - 9)/(4 - 18/10) + e_k$, which equals $360/22 + e_k$. Thus, $x \approx 16.36$. Computing $x \cdot u_k + e_k$ for each task τ_k, we obtain values of 16.636 for tasks $\tau_1, \ldots, \tau_4$, and 23.727 for $\tau_5, \ldots, \tau_8$. Hence, the two tasks with the highest values for $x \cdot u_k + e_k$ are τ_5 and τ_6. Using the execution costs and utilizations of τ_5 and τ_6 and 15 as the Λ^{th} execution cost, we obtain an improved value of 10.9 for x, which is more than 5 units less than the initial value. The set of tasks with the highest value for $x \cdot u_k + e_k$ is not altered in the next iteration, and so, the procedure terminates.

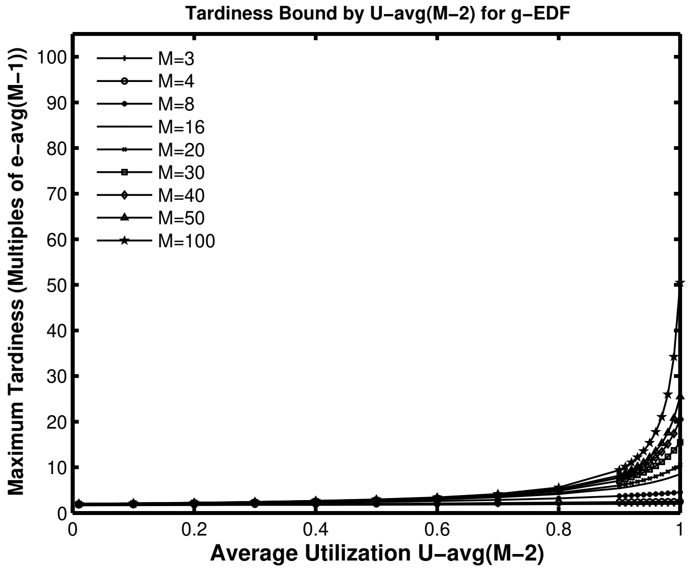
A similar procedure can be used for computing an improved bound for g-NP-EDF also, except that Λ tasks τ_k with the highest values for $x \cdot u_k + e_k$ have to be considered.

Improving computation time. Computing the tardiness bound or determining utilization restrictions on tasks for a given bound involves selecting $O(M)$ tasks with the highest utilizations, and $O(M)$ tasks with the highest execution costs (or the highest values for $e_i - b_i$), and summing these values. Because worst-case [38] and expected [58, 64] linear-time selection algorithms are known, these are $O(N+M)$ computations. If speed is a concern, as in online admission tests, and if $u_{\max}$ and $e_{\max}$ are known, then $O(1)$ computations, which assume a utilization of $u_{\max}$ and an execution cost of $e_{\max}$ for each of the $M-1$ tasks, can be used, at the expense of more pessimistic values. Under this assumption, tardiness bounds under g-EDF and g-NP-EDF are $((M-1)e_{\max} - e_{\min})/(M - (M-2)u_{\max}) + e_i$ and $(M \cdot e_{\max} - e_{\min})/(M - (M-1)u_{\max}) + e_i$, respectively, for each task τ_i. Similarly, for a tardiness bound of $x + e_i$ for every τ_i under g-EDF, it is sufficient that $u_{\max}$ be at most $(M \cdot x - (M-1)e_{\max} + e_{\min})/((M-2)x)$. We will refer to the bounds computed using $u_{\max}$ and $e_{\max}$ as EDF-FAST and NP-EDF-FAST, respectively.

Tightness of the bound. As discussed in the introduction, we do not believe that our results are tight for either g-EDF or g-NP-EDF. It can be verified that for both g-EDF and g-NP-EDF, the bounds derived are increasing functions of M. In addition, when $U_{sum} = M$, the g-EDF (resp., g-NP-EDF) bound is an increasing function of the average of the highest $M-2$ (resp., $M-1$) task utilizations, denoted $u_{\text{avg}}(M-2)$ (resp., $u_{\text{avg}}(M-1)$), and the average of the highest $M-1$ (resp., M) task execution costs, denoted $e_{\text{avg}}(M-1)$ (resp., $e_{\text{avg}}(M)$). Approximate plots of the tardiness bounds computed for g-EDF and g-NP-EDF for



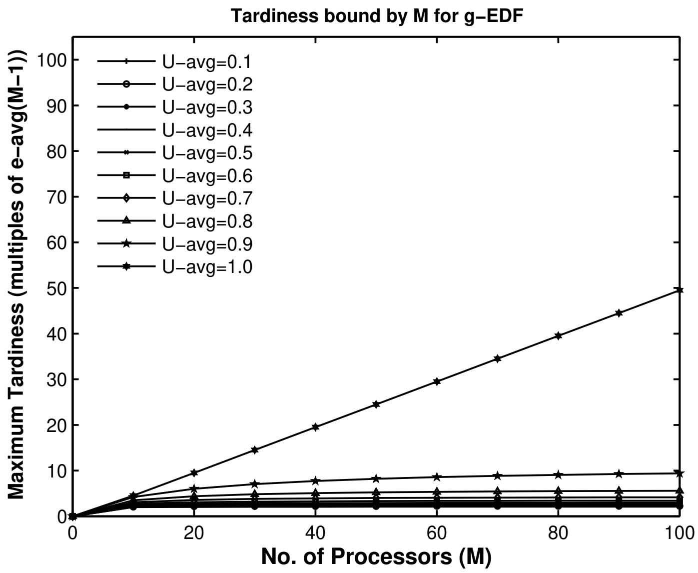
(a)



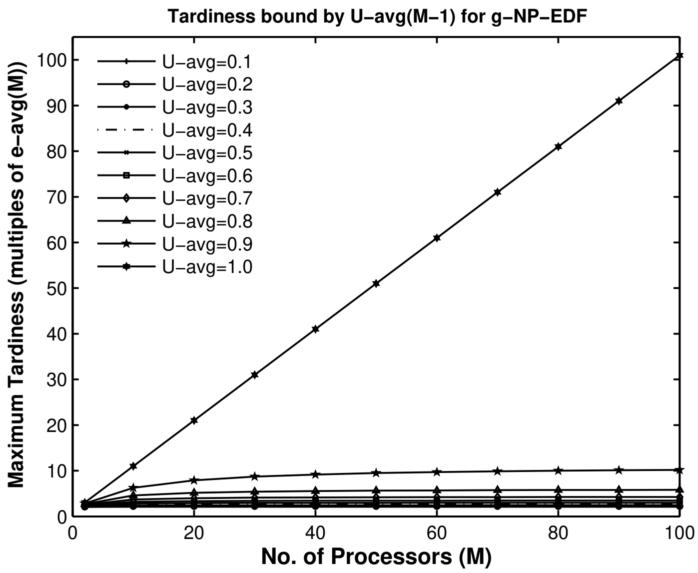
(b)

Figure 4.3: Tardiness bounds for **(a)** g-EDF and **(b)** g-NP-EDF for varying number of processors (M) as functions of average task utilization (u_{avg}). Values of M are higher for curves higher up.

varying values of M and u_{avg} are shown in Figures 4.3 and 4.4. The plotted bounds are in multiples of $e_{\text{avg}}(M-1)$ for g-EDF and $e_{\text{avg}}(M)$ for g-NP-EDF. As can be seen from the plots, the bounds are quite reasonable unless both M and u_{avg} are both high. For instance, for g-EDF, if u_{avg} is at most 0.75 (resp., 0.5), then the bound guaranteed is approximately at most



(a)



(b)

Figure 4.4: Tardiness bounds for **(a)** g-EDF and **(b)** g-NP-EDF for varying average task utilizations (u_{avg}) as functions of the number of processors M. u_{avg} is higher for curves higher up.

$5 \cdot e_{\text{avg}}$ (resp., $3 \cdot e_{\text{avg}}$) for all M, and if M is at most 8, then the bound is approximately at most $4 \cdot e_{\text{avg}}$ even when u_{avg} is close to 1.0. If $M = 8$, then the tardiness bound for g-EDF is at most $3 \cdot e_{\text{avg}}$ (resp., $2 \cdot e_{\text{avg}}$) when u_{avg} is at most 0.75 (resp., 0.5). Of course, whether that much tardiness is tolerable is application dependent. However, if execution costs are low, say of the order of a few milliseconds, as can be expected with modern processors, then in many cases, the tardiness bounds guaranteed can be expected to be within limits.

Though the bounds may not be tight, in general, we can show that our result for g-EDF is within approximately $e_{\max}$ of tardiness possible in the worst case for small values of $u_{\max}$. For this, consider a task system τ that consists of a primary task $\tau_1(e_1, p_1)$ and $(M - u_1)p_1/\delta$ auxiliary tasks. Let $\delta \ll e_1$ and p_1 be the execution cost and period, respectively, of each auxiliary task. Let δ divide $(M - u_1)$ evenly and let p_1 be a multiple of M. The total utilization of this task system is $(\delta/p_1) \times ((M - u_1)p_1/\delta) + u_1 = M$. Suppose that the first job of each task is released at time 0 and suppose that the first job of every auxiliary task is scheduled before $\tau_{1,1}$ and executes for a full δ time units. In such a schedule, the auxiliary jobs will execute continuously until time $(\frac{(M-u_1)p_1}{\delta} \times \delta)/M = p_1 - e_1/M$ on each processor. Hence, τ_1 will not begin executing until time $p_1 - e_1/M$, and hence would not complete until $p_1 + e_1 \cdot (1 - 1/M)$ for a tardiness of $e_1 \cdot (1 - 1/M) = e_1 \cdot ((M - 1)/M)$. By choosing $e_1 = M$, this tardiness can be made to equal $e_1 - 1$. In this example, $e_{\max} = e_1$, and hence tardiness is at least $e_{\max} - 1$. In the example schedule described above, tardiness is independent of $u_1 = e_1/p_1$, and hence, holds even when u_1 is arbitrarily small, which is possible by choosing $p_1 \gg e_1$. Note that when $u_{\max}$ is arbitrarily low, a tardiness bound under g-EDF computed using EDF-FAST is around $((M-1) \cdot e_{\max} - e_{\min})/M + e_{\max}$, which is $((M-1) \cdot M - \delta)/M + M$ for this example (because $e_{\max} = M$). This bound differs from the observed tardiness of $M - 1$ by $M - \delta/M$, which is at most $M = e_{\max}$. (The bound computed using EDF-BASIC is $(M + (M-3) \cdot \delta)/(M - u_1 - (M-3) \cdot u_1 \cdot \delta/M) + M$ and differs from the observed tardiness by $(M + (M-3) \cdot \delta)/(M - u_1 - (M-3) \cdot u_1 \cdot \delta/M) + 1$, which can be verified to be at most $(2 + \delta)M/(M - 1)$ for $u_1 \leq \frac{1}{1+\delta}$.)

Tightness of the g-EDF bound on two processors. By Theorem 4.2, tardiness is at most $(e_{\max} + e_\ell)/2$ for task τ_ℓ on two-processor systems. The following is an example that shows that this result is reasonably tight. Let $\tau_1(1, 2)$, $\tau_2(1, 2)$, and $\tau_3(2k+1, 2k+1)$, where $k \geq 1$, be three periodic tasks all of whose first jobs are released at time zero. If deadline ties are resolved in favor of τ_1 and τ_2 over τ_3, then on two processors, tardiness for jobs of τ_3 can be as high as $2k$ time units. (If τ_3 is favored, then its jobs can miss by $2k - 1$.) Such a schedule for a task system with $k = 7$ is shown in Figure 4.5. In this schedule, the sixth job of τ_3, with deadline at time 90, completes executing at time 104, and hence, has a tardiness of 14 time units, which is $e_{\max} - 1$. The schedule pattern in $[73, 104)$ repeats after time

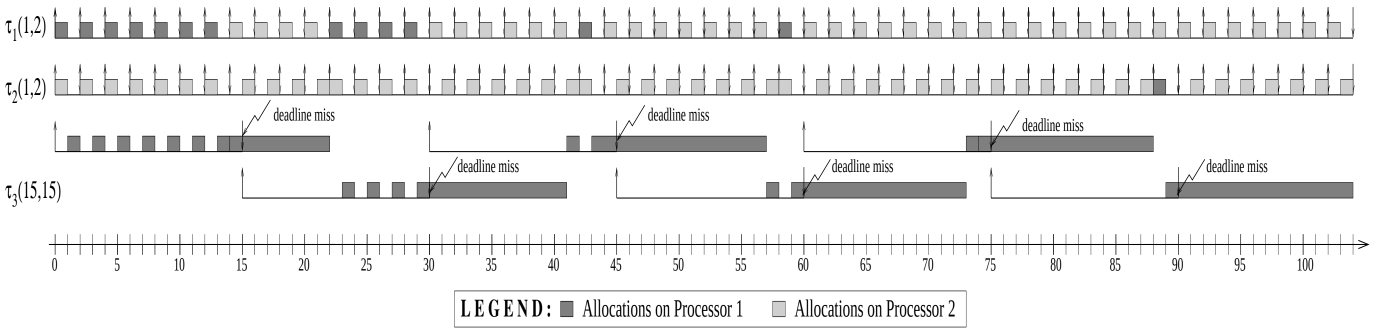


Figure 4.5: A schedule under g-EDF for the the first few jobs of the task system indicated on two processors. Tardiness for the sixth job of τ_3 is 14 time units, which is $e_{\max} - 1$.

104, and so tardiness converges to 14 time units after time 104. For this task set, estimated tardiness is $e_{\max} = 2k + 1$, and for large k, the difference between the estimated and actual tardiness is negligible. The above example can be tweaked to generate other task sets in which the maximum utilization is much less than unity, but which can yet miss their deadlines by $e_{\max} - 1$ time units under g-EDF. A task system with the following seven tasks is one example: $\tau_1(1, 4), \ldots, \tau_6(1, 4)$, $\tau_7(2k + 1, 4k + 2)$, where $k \geq 1$. In this example, τ_7 can miss deadlines by up to $2k$ time units, and the remaining tasks by up to k time units. (Note that $2k = e_{\max} - 1$ and $k = \frac{e_{\max}+e_\ell}{2} - 1$, for $1 \leq \ell \leq 6$.)

An empirical evaluation of the accuracy of the bounds is provided in Section 4.5. Interestingly, we have found that tardiness under g-EDF can exceed $e_{\max}$ even for task systems with $u_{\max}$ near 1/2. The following is one such task set: $\tau_1(1, 2), \ldots, \tau_4(1, 2)$, $\tau_5(1, 5), \ldots, \tau_7(1, 5)$, $\tau_8(1, 11)$, $\tau_9(34, 110)$, $\tau_{10}(23, 63)$, $\tau_{11}(7, 18)$,$\tau_{12}(7, 18)$, $\tau_{13}(3, 7)$,$\tau_{14}(3, 7)$. The total utilization of this task set is five. When scheduled on five processors with deadline ties resolved using task indices, a job of τ_9 misses its deadline at time 7295 by $35 > 34 = e_{\max}$ time units. (The EDF-BASIC and EDF-ITER bounds for this task set are 54 and 51.78, respectively.) Hence, the best bound possible for an arbitrary task set definitely exceeds $e_{\max}$.

4.4 A Useful Task Model Extension

In this section, we consider a useful extension to the sporadic task model, and discuss how a tardiness bound can be derived under it. We argue that the derivation of such a bound involves only minimal changes to that used for the base model (*i.e.*, the model described in Section 4.2).

As explained later, the extension that we consider can be used to model certain dynamic task behaviors. In this extended model, the number of tasks associated with the task system is allowed to exceed N and the total utilization of all tasks is allowed to exceed M. (The number of tasks could potentially be infinite.) However, to prevent overload, at any given time, only a subset of tasks whose total utilization is at most M is allowed to be *active*,[5] *i.e.*, is allowed to release jobs. Additionally, we allow the final job of a task to *halt* at some time t_h before its deadline, provided that the allocation that the job receives in the actual schedule is at most the allocation it receives up to t_h in a PS schedule and the job is not executing in a non-preemptive segment at t_h. When a job halts, its execution cost is altered to equal the amount of time that the job actually executed for in the actual schedule up to t_h. Hence, the halting job receives no allocation in the PS schedule after t_h, even if its deadline is after t_h. At any time t, each task τ_i can be in one of the following states.

- *Active*, if the first job of τ_i is released at or before t, the deadline of its final job is after t, and its final job has not halted at or before t. A task whose final job has its deadline at or before t is not considered active at t even if the final job is pending at t.
- *Inactive*, if the release time of the first job of τ_i is after t.
- *Terminated*, if the deadline of τ_i's final job is before t.
- *Halted*, if the final job has halted but its deadline has not elapsed.

As can be easily seen, a task that is either inactive or terminated at time t cannot have active jobs at t. Also, though the deadline of a halted job may be after its halting time, the job receives no allocation in the PS schedule after it halts.

The set of all tasks associated with the task system is partitioned into N task classes such that the following hold: **(i)** active intervals are disjoint for every two tasks in each class and **(ii)** tasks within a class are governed by precedence constraints, *i.e.*, the first job of a task cannot begin execution until all jobs of all tasks with earlier active intervals in its class have completed execution. The second requirement implies that tasks that are not bound by precedence constraints should belong to different classes even if their active intervals are disjoint.

[5] The term active is overloaded in that it was defined previously for jobs (see Definition 4.1 in Section 4.3.1). The definition of this term is different for tasks, and active tasks are not to be confused with active jobs. Specifically, a task can be considered to be active at some time even if it has no active job at that time.

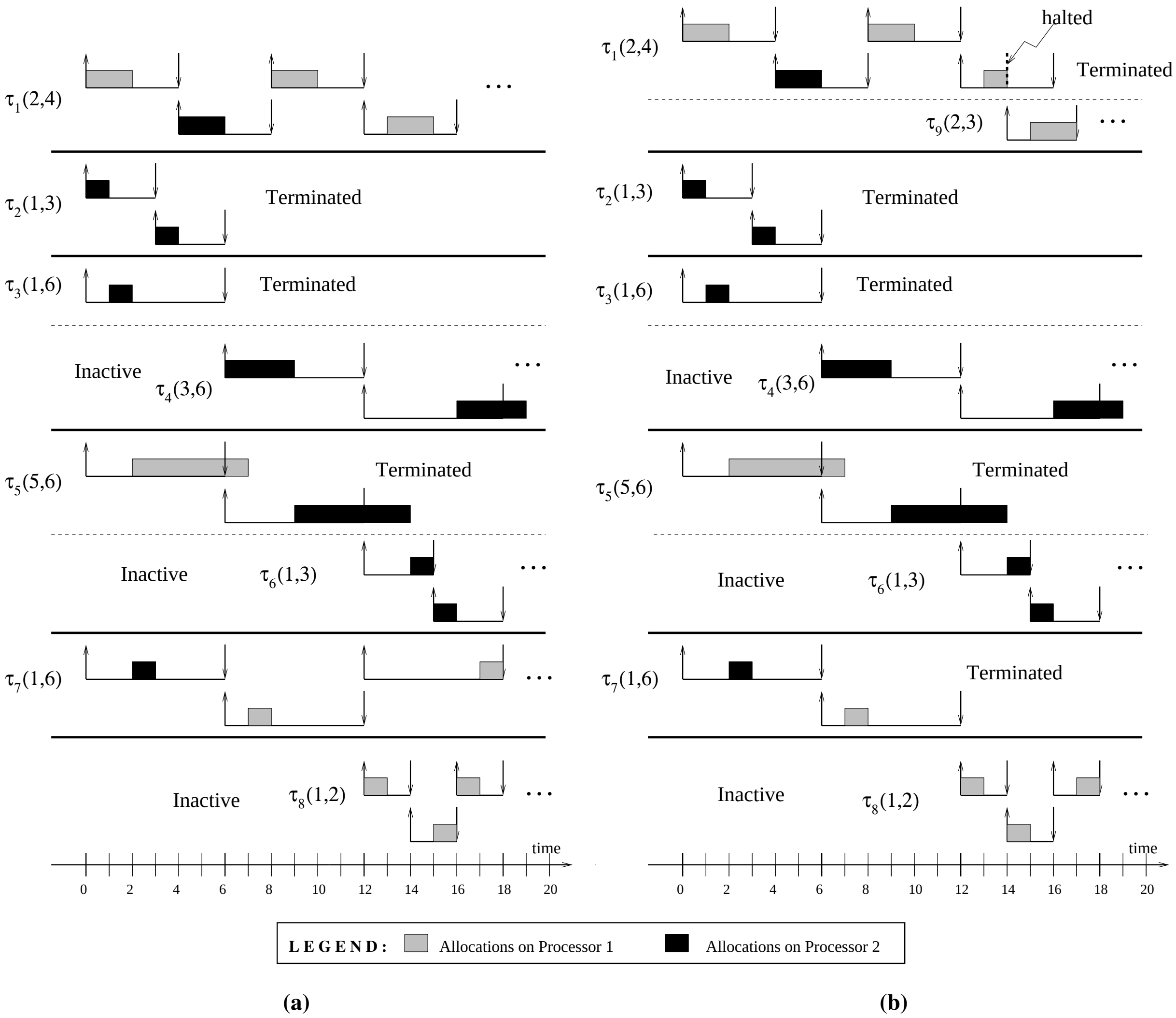


Figure 4.6: Example g-EDF schedules for extended sporadic task systems. **(a)** No job halts. **(b)** The fourth job of τ_1 halts at time 14.

Examples of task systems that conform to the proposed extension are shown in Figure 4.6. In each example, tasks that belong to different classes are demarcated using solid lines; those within a class are separated using dashed lines. An initial few jobs are shown for each task. There are eight tasks and six task classes in the task system in inset (a). Each task becomes active at the release time of its first job. Tasks whose jobs are followed by ellipsis are still active at time 18 and may be active beyond that time. Termination times for the remaining tasks are given by the deadlines of their final jobs. For example, τ_2 and τ_3 terminate at time 6, and τ_6 is active in the interval $[12, \infty)$. It can be verified that the total utilization of all the active tasks at any instant is 2. Note that since τ_5 and τ_6 are in the same class, $\tau_{6,1}$ cannot

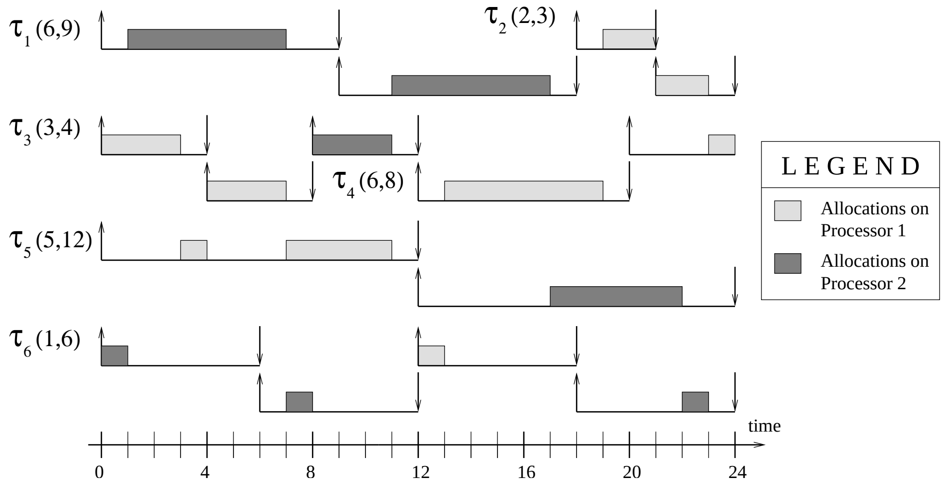


Figure 4.7: Example application of the extended task model to tasks with variable per-job execution costs.

begin execution until both jobs of τ_5 have completed executing. Hence, at time 13, the fourth job of τ_1 is scheduled even though its deadline (which is at time 16) is later than that of $\tau_{6,1}$ (which is at time 15). Inset (b) shows an example wherein a job is halted before its deadline. In this example, at time 14, the fourth job of τ_1, $\tau_{1,4}$, is halted, and a new task, τ_9, which belongs to the same class as τ_1, becomes active. At time 14, $\tau_{1,4}$ receives equal allocations (of one time unit) in both the g-EDF and PS schedules, and hence, halting is permissible.

Applications. The extended task model can readily be used to model tasks whose job execution costs vary but whose utilizations remain unchanged. An example is shown in Figure 4.7. The task system in this example has two tasks with fixed per-job execution costs and two with variable execution costs. The first variable execution-cost task has a utilization of 2/3, an execution requirement of six for its first two jobs, and two, for later jobs. The second such task has a utilization of 3/4, an execution requirement of three for its first three jobs, and six, for later jobs. Both the tasks can be represented using task classes of the extended task model. In Figure 4.7, tasks τ_1 and τ_2 belong to the first task class. τ_1 is active in the interval $[0, 18)$ and τ_2 becomes active at time 18. Similarly, tasks τ_3 and τ_4 together model the second variable execution-cost task.

The extended task model can also be used with dynamic tasks whose utilizations can vary with time. In particular, a single task with a varying utilization can be modeled as a class of potentially infinite tasks, each with a non-changing utilization. Such an approach has been considered for scheduling dynamic task systems under g-EDF and g-NP-EDF in [37].

Tardiness bounds. The approach used for the base model can be used to derive a tardiness bound for the extended model if each task class of the extended model is treated as a distinct task of the base model. For clarity, we will initially assume that no job halts. Letting τ_ℓ^C denote task class ℓ (the superscript C is used to distinguish a task from a task class), in Lemma 4.6, the lag for τ_ℓ^C can easily be shown to be at most $x \cdot \max_{\tau_i \in \tau_\ell^C} u_i + e_{k,q}$, where $\tau_{k,q}$ is the earliest pending job of any of the tasks in τ_ℓ^C. Note that in a non-blocking, non-busy interval with continually increasing LAG, as considered in Lemma 4.9, at most Λ task classes can have pending jobs. Letting $u_\ell^C \stackrel{\text{def}}{=} \max_{\tau_i \in \tau_\ell^C} u_i$ and $e_\ell^C \stackrel{\text{def}}{=} \max_{\tau_i \in \tau_\ell^C} e_\ell$, it can be verified that Lemmas 4.9 and 4.10 directly apply to the extended model (*i.e.*, hold word for word) if each task class is considered as a task and a C superscript is added to the various terms. (μ_i^C and ϵ_i^C for all $1 \leq \imath \leq N$ are defined for task classes in a straightforward manner). Hence, the tardiness bounds derived hold if the maximum utilization and execution cost of each task class are used instead of individual task utilizations and execution costs.

We now discuss why halting is not of issue. Recall that a job can halt only if the allocation to it in the PS schedule is no less than the allocation it receives in the actual schedule. Further, the actual execution cost of the halting job is altered to equal the allocation the job receives in the actual schedule. Hence, the allocation to the job in the PS schedule can also retroactively be altered to equal its actual execution cost. Further, since the job will receive no allocation in the PS schedule after it halts, it is safe to reclaim the utilization of the halting task and activate some other task. That is, there is no risk of overload (*i.e.*, U_{sum} will not exceed M), which can adversely impact other tasks, in the interval spanning the job halting time and job deadline. Finally, by definition, a job can halt only before its deadline and is not pending when it halts. Therefore, because allocations in the PS schedule are altered as described above, the tardiness and lag of a job are zero when the job halts. Hence, lag bounds derived for task classes are not worsened even if its jobs halt. Also, since a job cannot halt in a non-preemptive segment, bounds on blocking times are not altered.

4.5 Simulation-Based Evaluation

In this section, we describe the results of two sets of simulation experiments conducted using randomly-generated task sets to evaluate the accuracy of the tardiness bounds for g-EDF and g-NP-EDF derived in Section 4.3.

The first set of experiments compares the tardiness bounds given by EDF-BASIC and NP-

EDF-BASIC and their fast (EDF-FAST and NP-EDF-FAST) and iterative (EDF-ITER and NP-EDF-ITER) variants for 4 and 8 processors (M). For each M, 10^6 task sets were generated. For each task set, new tasks were added as long as U_{sum} was less than M and a final task was added so that U_{sum} equalled M. For each task τ_i, first its utilization u_i was generated as a uniform random number between $(0, y]$, where y was fixed at 0.1 for the first $100,000$ task sets, and was incremented in steps of 0.1 for every $100,000$ task sets. τ_i's execution cost was chosen as a uniform random number in the range $(0, 20]$. For each task set, the maximum tardiness of any task and the average of the maximum tardiness of all tasks as given by the six bounds (three each for g-EDF and g-NP-EDF) were computed. The mean maximum tardiness is plotted for $M = 4$ and $M = 8$ in Figures 4.8 and 4.9 as a function of e_{avg} and u_{avg}, respectively, where u_{avg} denotes the average of the $M - 2$ highest utilizations and e_{avg} that of the $M - 1$ highest execution costs. (Mean average tardiness is around $e_{\text{avg}}/2$ time units less.) The descriptions of the plots can be found in the caption of the figure. The rest of this section discusses the results in some detail.

Comparison of BASIC and FAST. Referring to Figures 4.8(a) and (c) and Figure 4.9(a), for $M = 4$, the difference between BASIC and FAST is negligible for g-EDF, but is quite considerable for g-NP-EDF at high values of u_{avg} and e_{avg}. This is due to an additional $e_{\max}$ term in the numerator and a negative $u_{\max}$ term in the denominator of NP-EDF-FAST. The difference widens with M for both g-EDF and g-NP-EDF for high u_{avg}. For $M = 8$, as can be seen in Figures 4.8(c) and 4.9(b), the NP-EDF-FAST bound can be up to twice as much as that of NP-EDF-BASIC. For g-EDF, the difference seems to be tolerable at $M = 8$, but can be expected to be quite significant for higher values of M. Overall, FAST appears to yield good results for small M and small u_{avg} or e_{avg}.

Comparison of BASIC and ITER. The difference between ITER and BASIC is almost the same as that between BASIC and FAST for g-EDF for $M = 4$ (refer to Figures 4.8(a), 4.8(b), and 4.9(a)), but is lower for g-NP-EDF. The difference increases with increasing M for both g-EDF and g-NP-EDF. This is because there is not much increase in ITER with increasing M.

Comparison of g-EDF and g-NP-EDF. While there is a large difference between the FAST versions of g-EDF and g-NP-EDF, which widens with increasing M, the difference between EDF-ITER and NP-EDF-ITER is much less and narrows with increasing M. The peak difference between the ITER versions, which is less than 20 time units, occurs for $M = 4$, $0.9 < u_{\text{avg}} \leq 1.0$,

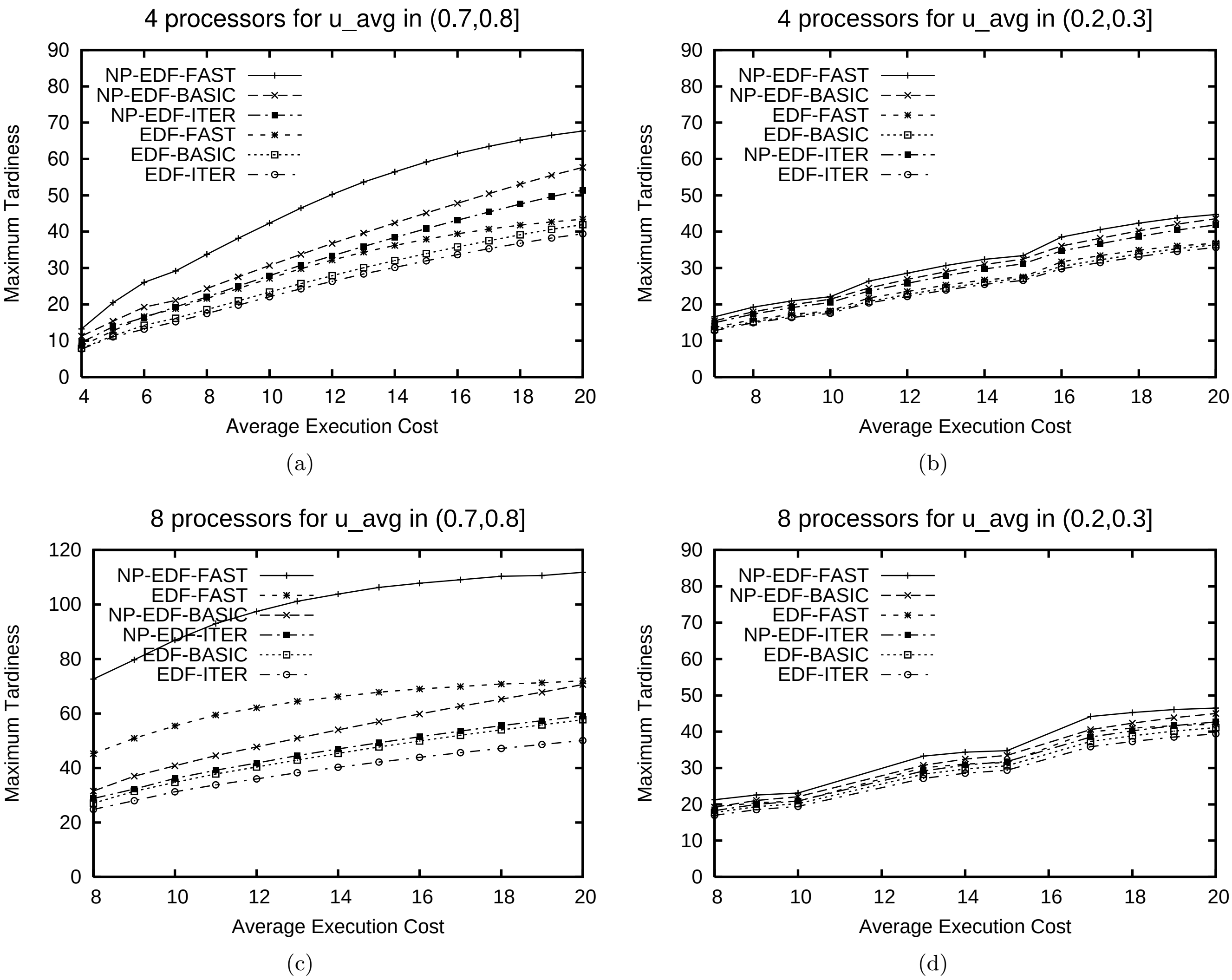
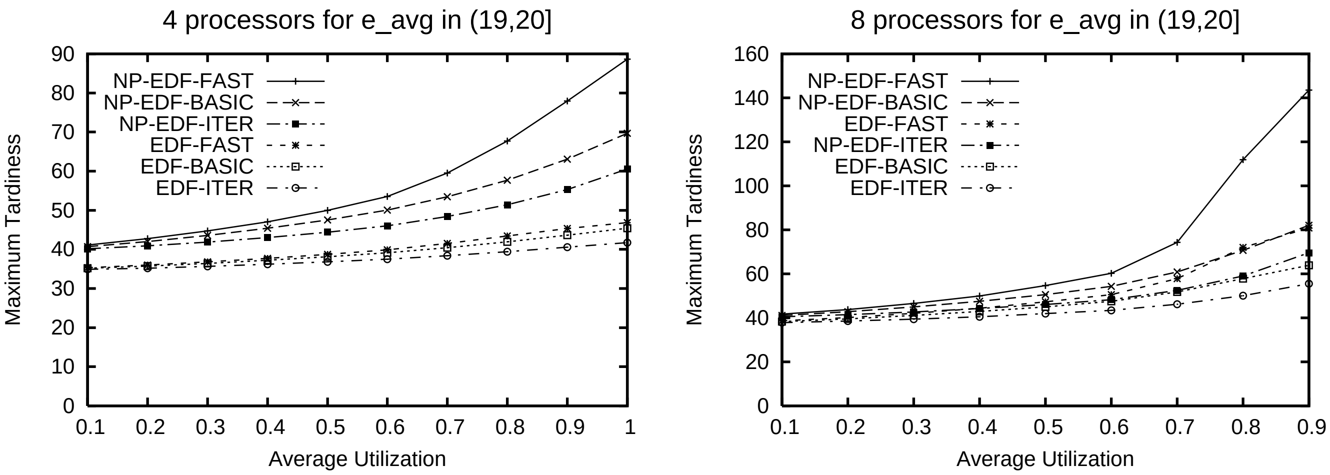


Figure 4.8: Comparison of the three tardiness bounds derived for g-EDF and g-NP-EDF. Tardiness bounds are plotted as functions of e_{avg} for **(a) & (b)** $M = 4$ and **(c) & (d)** $M = 8$ processors. u_{avg} is in $(0.7, 0.8]$ in **(a) & (c)** and is in $(0.2, 0.3]$ in **(b) & (d)**. The order of the legends and curves coincide in all the graphs.

and $19 < e_{\text{avg}} \leq 20$ (see Figure 4.9(a)). At low utilizations, the difference is around five time units for $M = 4$ and three time units for $M = 8$. The difference between the BASIC versions is also not much and decreases with M. However, it should be noted that while these observations hold for the task sets generated here, they cannot be taken to be conclusive. Another point worth mentioning is that, these results assume the same worst-case execution costs for both g-EDF and g-NP-EDF, whereas in practice the estimates for g-NP-EDF will be lower due to the absence of job preemptions and migrations. This should further close the gap between the two.



(a) (b)

Figure 4.9: Tardiness bound as a function of u_{avg} for e_{avg} in $(19, 20]$ on **(a)** $M = 4$ and **(b)** $M = 8$ processors. (The order of the legends and curves coincide in both the graphs.)

Though not shown here, we also plotted the tardiness bounds by u_{avg} for low and medium values of e_{avg}, and found that in comparison to the plots in Figure 4.9, all the bounds are proportionately reduced.

Comparison of ITER to actual tardiness. The experiments in the second set compare the bounds estimated by EDF-ITER and NP-EDF-ITER, the best bounds derived, to actual tardiness observed under g-EDF and g-NP-EDF, respectively. In this case, 100,000 task sets were generated for each M. For each task set, the tardiness bounds given by EDF-ITER and NP-EDF-ITER were computed. Also, a g-EDF and a g-NP-EDF schedule were generated for each task set for 20,000 and 50,000 time units, respectively, and the maximum tardiness observed in each schedule was noted. Plots of the average of the estimated and observed values for task sets grouped by e_{avg} and u_{avg} are shown in Figure 4.10. For medium values of e_{avg}, the estimates are twice as much as the observed values, with the difference increasing with increasing execution costs. It is somewhat surprising that actual tardiness does not increase much with increasing e_{avg} and that it decreases in some cases. It should also be noted that the difference is higher when M is higher. These plots are a clear indication that there is significant room for improvement. However, tightening the bounds any further does not seem to be easy.

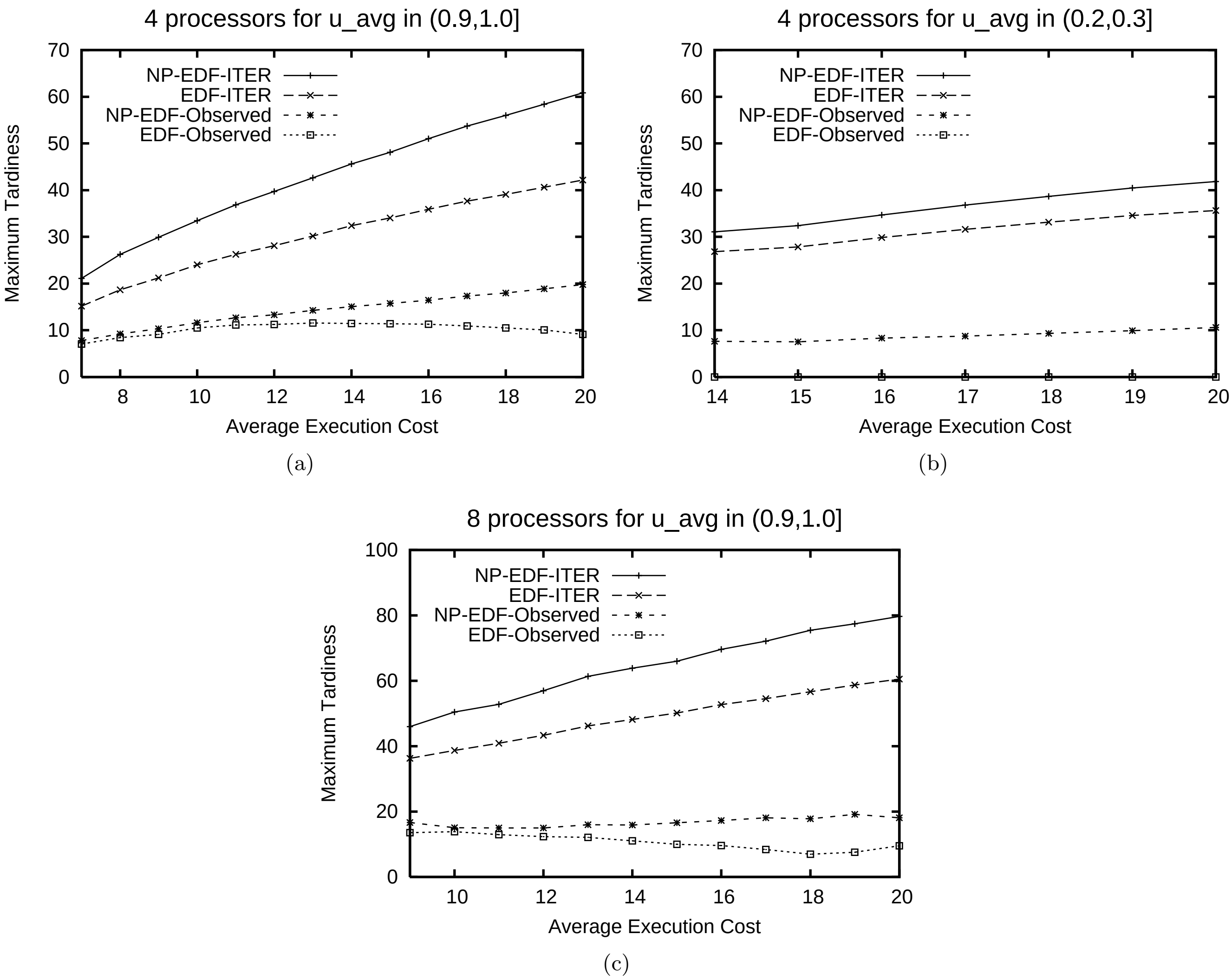


Figure 4.10: Comparison of EDF-ITER and NP-EDF-ITER to tardiness observed in actual g-EDF and g-NP-EDF schedules for **(a) & (b)** $M = 4$, and for **(c)** $M = 8$.

4.6 Summary

We have derived tardiness bounds under preemptive and non-preemptive global EDF for sporadic real-time task systems on multiprocessors, when the total utilization of a task system is not restricted, but may equal the number of processors, M. Our tardiness bounds depend on the total system utilization, and per-task utilizations and execution costs — the lower these values, the lower the tardiness bounds. Though the bounds are not believed to be tight, they are reasonable and should be acceptable as long as the maximum task utilization is not very high, say above 0.8. These results should help in improving the effective system utilization when soft real-time tasks that can tolerate bounded deadline misses but require guarantees on the long-run fraction of the processing time allocated are multiplexed on a multiprocessor. Improving the accuracy of the bounds remains a challenging open problem.

Our task model can alternatively be viewed as one in which the relative deadline of each task is greater than its period by an amount that is the same for all tasks and is dependent on the parameters of the task system. Our conditions that check if a tardiness bound can be guaranteed can then be used as schedulability tests for such task systems. While it is possible to extend the EDF schedulability tests derived in prior research discussed in Section 4.1 to task systems with relative deadlines greater than periods, it does not seem likely that such extensions would allow total utilization to equal M even when per-task utilizations are low and relative deadlines are large.

Apart from not being tight, another limitation of our result is that the tardiness bound that can be guaranteed to each task is fixed and is dependent on task parameters. Guaranteeing arbitrary and different tardiness bounds to different tasks is another challenging problem towards which we have taken some steps in later work [52].

Chapter 5

EDF-fm: A Restricted-Migration Algorithm for Soft Real-Time Systems[1]

In the previous chapter, we showed that on $M \geq 2$ processors, preemptive and non-preemptive global EDF can guarantee bounded tardiness to every sporadic task system τ with $U_{sum}(\tau) \leq M$. We also explained that since run-time overheads are lower under global EDF than under known optimal algorithms, effective system utilization can be improved by scheduling soft real-time systems under global EDF. However, task migrations under global EDF are still unrestricted, which may be unappealing for some systems, but if migrations are forbidden entirely, then bounded tardiness cannot be guaranteed. In this chapter, we address the issue of finding an acceptable middle ground.

Our contributions are twofold. First, we present an algorithm called EDF-fm, which treads a middle path between full-migration and no-migration algorithms, by restricting, but not eliminating, task migrations, and derive a tardiness bound that can be guaranteed under it. The tardiness bound derived can be computed in $\mathcal{O}(N)$ time, where N is the number of tasks. Though our algorithm adheres to the conditions of the middle entry of Table 1.1 (restricted migration, restricted dynamic priorities), the degree of migration that is needed is in fact lower than that suggested by that entry: under EDF-fm, only up to $M-1$ tasks, where M is the number of processors, *ever* migrate, and those that do, do so only between jobs and only between two processors. As noted in [42], migrations between jobs should not be much of a concern for tasks for which little state is carried over from one job to the next.

[1]Contents of this chapter previously appeared in preliminary form in the following paper:
[10] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, July 2005.

The maximum tardiness that any task may experience under EDF-fm is dependent on the per-task utilization cap assumed—the lower the cap, the lower the tardiness threshold. Even with a cap as high as 0.5 (*half* of the capacity of one processor), bounded tardiness can be guaranteed for *all* the task systems. (In contrast, if $\alpha = 0.5$ in the middle entry of Table 1.1, then approximately 50% of the system's overall capacity may be lost.) The percentage of task systems for which bounded tardiness can be guaranteed decreases with the per-task utilization cap and the number of processors. In our simulation experiments, we found that on sixteen (resp., four) processors, bounded tardiness could be guaranteed to roughly 50% (resp., 75%) and 75% (resp., 95%) of the task sets when $u_{\max} = 1.0$ and $u_{\max} = 0.8$, respectively. Hence, our algorithm should enable a wide range of soft real-time applications to be scheduled in practice with *no constraints on total utilization*. In addition, when a job misses its deadline, we do *not* require a commensurate delay in the release of the next job of the same task. As a result, each task's required processor share is maintained over the long term.

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

The rest of this chapter is organized as follows. In Section 5.1, Algorithm EDF-fm is presented and a tardiness bound under it that can be computed in $\mathcal{O}(N)$ time is derived. Techniques and heuristics that can be used to reduce tardiness observed in practice, and exponential-time iterative formulas for computing more accurate tardiness bounds, as described above, are presented in Section 5.2. Then, in Section 5.3, a simulation-based evaluation of our basic algorithm and proposed heuristics is presented, and the accuracy of the tardiness bound derived is assessed. Section 5.4 concludes.

5.1 Algorithm EDF-fm

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

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

$s_{i,j}$	$\stackrel{\text{def}}{=}$	Percentage of P_j's processing capacity (expressed as a fraction) allocated to τ_i, $1 \leq i \leq N, 1 \leq j \leq M$. ($\tau_{i,j}$ is said to have a *share* of $s_{i,j}$ on P_j.)	(5.1)
$f_{i,j}$	$\stackrel{\text{def}}{=}$	$\frac{s_{i,j}}{u_i}$, the fraction of τ_i's total workload that P_j can handle, $1 \leq i \leq N,\ 1 \leq j \leq M$.	(5.2)
ρ_i	$\stackrel{\text{def}}{=}$	Maximum percentage of P_i's processing capacity (expressed as a fraction) that can be allocated to tasks in τ, $1 \leq i \leq N$. (In other words, the sum of all shares assigned to tasks on P_i may not exceed ρ_i.)	

5.1.1 Assignment Phase

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

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

Algorithm Assign-Tasks()
    local var
    ▷ identifier for current processor
        proc : 1..M initially 1;
    ▷ identifier for current task
        task : 1..N;
    ▷ unassigned utilization on proc
        AvailUtil : rational;

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

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

Figure 5.1: Algorithm Assign-Tasks.

In Figure 5.1, pseudo-code is given for a task-assignment algorithm, denoted Assign-Tasks, that satisfies the following properties for every task system τ with $u_{\max}(\tau) \leq \min_{1 \leq \imath \leq N} \rho_i$ and $U_{sum}(\tau) \leq \sum_{i=1}^{M} \rho_i$.

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

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

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

In this pseudo-code, the i^{th} element $u[i]$ of the global array u represents the utilization u_i of task τ_i, $s[i][j]$ denotes $s_{i,j}$ (as defined in (5.1)), the i^{th} element of array p, which is array $p[i]$, contains the processor(s) to which task i is assigned; arrays $m[i]$ and $f[i]$ denote the migrating tasks and fixed tasks assigned to Processor i, respectively. Note that $p[i]$ and $m[i]$ are each vectors of size two.

ASSIGN-TASKS assigns tasks in sequence to processors, starting from the first processor. Tasks and processors are both considered sequentially. Local variables *proc* and *task* denote the current processor and task, respectively. Tasks are assigned to *proc* as long as the upper limit, ρ_{proc}, on the processing capacity of *proc* is not exhausted. If the current task *task* cannot receive its full share of u_{task} from *proc*, then part of the processing capacity that it requires is allocated on the next processor, $proc + 1$, such that the sum of the shares allocated to *task* on the two processors equals u_{task}. Note that if $u_{\max} \leq \rho_i$, for all i, such an assignment is possible for any task. It is easy to see that if $U_{sum} \leq \sum_{i=1}^{M} \rho_i$ also holds, then assigning tasks to processors following this simple approach satisfies (P1)–(P3).

Example 5.1. Consider a task set τ with the following nine tasks: $\tau_1(5,20)$, $\tau_2(3,10)$, $\tau_3(1,2)$, $\tau_4(2,5)$, $\tau_5(2,5)$, $\tau_6(1,10)$, $\tau_7(2,5)$, $\tau_8(7,20)$, and $\tau_9(3,10)$. The total utilization of this task set is three. A share assignment produced by ASSIGN-TASKS when $\rho_1 = \rho_2 = \rho_3 = 1.0$ is shown in Figure 5.2. In this assignment, τ_3 and τ_7 are migrating tasks; the remaining tasks are fixed. τ_3 has a share of $\frac{9}{20}$ on processor P_1 and a share of $\frac{1}{20}$ on processor P_2, while τ_7 has shares of $\frac{1}{20}$ and $\frac{7}{20}$ on processors P_2 and P_3, respectively.

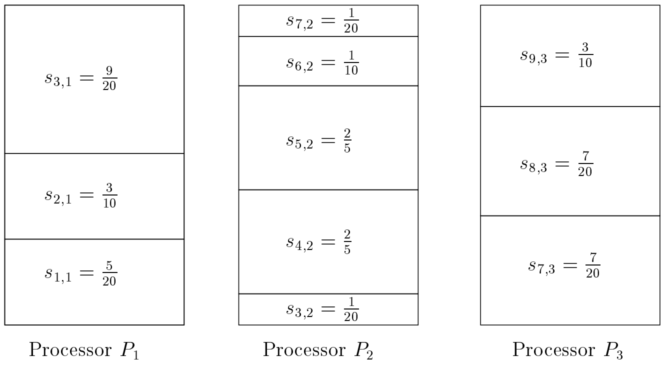


Figure 5.2: Task assignment on three processors for the task set in Example 5.1 using Algorithm ASSIGN-TASKS.

5.1.2 Execution Phase

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

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

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

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

Let us look at a concrete example that reveals this linkage among processors. Consider the task set τ, introduced earlier, with task assignments and processor shares shown in Figure 5.2. For simplicity, assume that the execution of the jobs of a migrating task alternate between the two processors to which the task is assigned. τ_3 releases its first job on P_1, while τ_7 releases its first job on P_3. (We are assuming such a naïve assignment pattern to illustrate the processor linkage using a short segment of a real schedule. Such a linkage occurs even with an intelligent job-assignment pattern if migrating tasks miss their deadlines.) A complete schedule up to time 27, with the jobs assigned to each processor scheduled using EDF, is shown in Figure 5.3.

In Figure 5.3, the sixth job of the migrating task τ_3 misses its deadline (at time 12) on P_2 and completes executing at time 14. This prevents the next job of τ_3 released on P_1 from being scheduled until time 14 and it misses its deadline. Because job releases are not postponed due to deadline misses, the seventh job of τ_3 is released at time 12 and has a deadline at time 14.

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

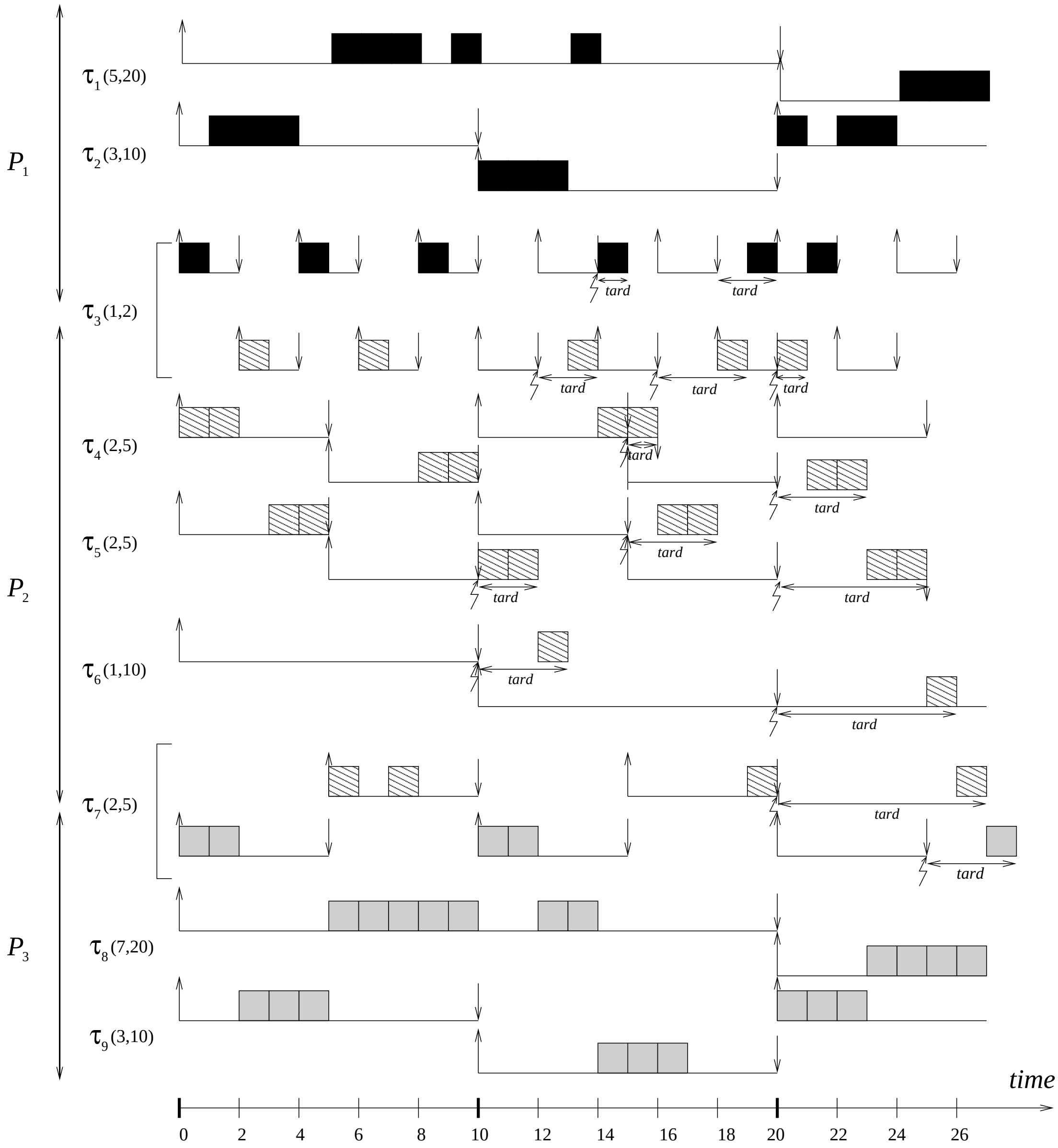


Figure 5.3: Illustration of processor linkage.

turn reduces the interval over which the fifth job of the same task can execute on P_3. Note that jobs $\tau_{9,3}$ and $\tau_{8,2}$ execute before $\tau_{7,5}$, as this job is forced to wait for its predecessor to complete executing on P_2. Thus, a nontrivial linkage is established among the processors that impacts system tardiness.

Per-processor scheduling rules. EDF-fm eliminates this linkage among processors by ensuring that *migrating tasks do not miss their deadlines*. Jobs of migrating tasks are assigned to processors using static rules that are independent of run-time dynamics. The jobs assigned

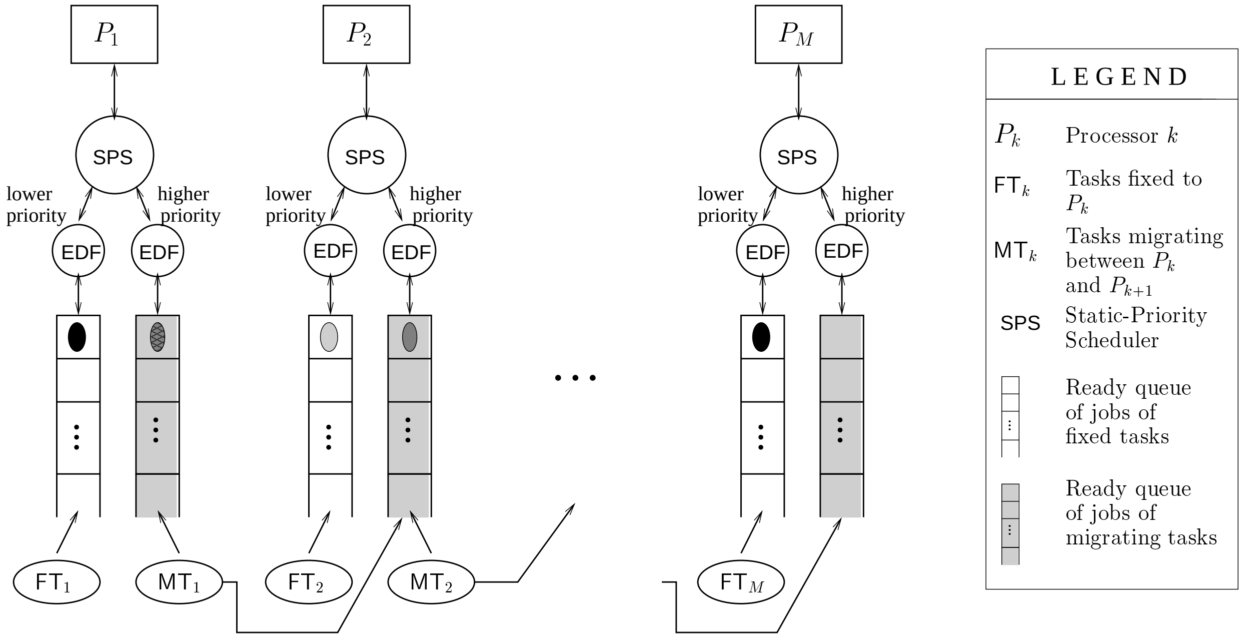


Figure 5.4: Schematic representation of EDF-fm in the execution phase.

to a processor are scheduled independently of other processors, and on each processor, migrating tasks are statically prioritized over fixed tasks. Jobs within each task class are scheduled using EDF, which is optimal on uniprocessors. A schematic representation of EDF-fm in the execution phase is shown in Figure 5.4. This priority scheme, together with the restriction that migrating tasks have utilizations at most 1/2, and the task assignment property (from (P2)) that there are at most two migrating tasks per processor, ensures that migrating tasks never miss their deadlines. Therefore, the jobs of migrating tasks executing on different processors do not impact one another, and each processor can be analyzed independently. Thus, the multiprocessor scheduling analysis problem at hand is transformed into a simpler uniprocessor one.

In the description of EDF-fm, we are left with defining rules that map the jobs of migrating tasks to processors. A naïve assignment of the jobs of a migrating task to its processors can cause an over-utilization on one of its assigned processors. EDF-fm follows a job assignment pattern that prevents over-utilization in the long run by ensuring that over well-defined time intervals (explained later), on each processor, the demand due to a migrating task is in accordance with its allocated share on that processor.

For example, consider the migrating task $\tau_7(2,5)$ in the example above. τ_7 has a share of $s_{7,2} = \frac{1}{20}$ on P_2 and $s_{7,3} = \frac{7}{20}$ on P_3. Also, $f_{7,2} = \frac{s_{7,2}}{u_7} = \frac{1}{8}$ and $f_{7,3} = \frac{s_{7,3}}{u_7} = \frac{7}{8}$, which imply

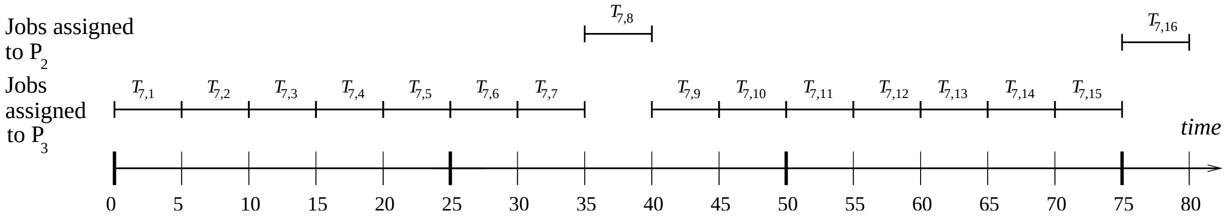


Figure 5.5: Assignment of periodically released jobs of migrating task τ_7 to processors P_2 and P_3.

that P_2 and P_3 should be capable of executing $\frac{1}{8}$ and $\frac{7}{8}$ of the workload of τ_7, respectively. Our goal is to devise a job assignment pattern that would ensure that, in the long run, the fraction of a migrating task τ_i's workload executed on P_j is close to $f_{i,j}$, and at any time, deviation from this ideal is minimized. One such job assignment pattern for τ_7 over interval $[0, 80)$ is shown in Figure 5.5. Assuming that τ_7 is a synchronous, periodic task,[2] the pattern in $[0, 40)$ would repeat every 40 time units.

In the job assignment of Figure 5.5, exactly one job out of every eight consecutive jobs of τ_7 released in the interval $[5k, 5(k+8))$, where $k \geq 0$, is assigned to P_2. Because $e_7 = 2$, τ_7 places a demand for two units of time, *i.e.*, $1/20^{\text{th}}$ of P_2, in $[5k, 5(k+8))$. Because τ_7 is allocated a share of $s_{7,2} = 1/20$ on P_2, this job assignment pattern ensures that in the long run τ_7 does not overload P_2. However, the demand due to τ_7 on P_2 over short intervals may exceed or fall below the share allocated to it. For instance, under the above job assignment, τ_7 requires two units of time, *i.e.*, $2/5$ of P_2 in the interval $[40k + 35, 40(k+1))$, and zero time units in the interval $[40k, 40k + 35)$. Similarly, exactly seven out of every eight consecutive jobs of τ_7 are assigned to P_3. Thus, τ_7 requires 14 units of time, or $7/20$ of the time, in $[5k, 5(k+8))$, and the share allocated to it matches this need. However, as with P_2, the demand due to τ_7 on P_3 over shorter intervals may deviate from its long-term share.

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

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

which fixed tasks can miss their deadlines due to the transient overload of migrating tasks is bounded.

A job assignment pattern similar to the one in Figure 5.5 can be defined for any migrating task. We draw upon some concepts of Pfair scheduling, described in Chapter 3, to derive formulas that can be used to determine such a pattern at run-time. Hence, before proceeding further, a brief digression on Pfair scheduling that reviews needed concepts is in order. (We stress that we are *not* using Pfair algorithms in our scheduling approach. We merely wish to borrow some relevant formulas from the Pfair scheduling literature.)

5.1.2.1 Digression: Review of Needed Pfair Scheduling Concepts

As discussed in Chapter 3, currently, Pfair scheduling [25] is the only known way of *optimally* scheduling recurrent real-time task systems on multiprocessors. Pfair algorithms achieve optimality by requiring each task to execute at a more uniform rate, given by its utilization, than mandated by the periodic or the sporadic task models. In fact, the allocation error at any time for optimal Pfair algorithms, in comparison to ideal fluid algorithms that can execute each task at its precise rate, is less than one time unit. It is also known that, in general, an allocation error lower than that guaranteed by Pfair, is not possible in practice [25]. Hence, we consider Pfair scheduling rules to be appropriate for distributing the jobs of migrating tasks.

Recall that in Pfair scheduling terminology, each task T has an integer execution cost $T.e$, an integer period $T.p \geq T.e$, and a rational weight, $wt(T) \stackrel{\text{def}}{=} T.e/T.p$. (As mentioned in Chapter 3, in the context of Pfair scheduling, tasks are denoted using upper-case letters without subscripts.)

Recall also that under Pfair scheduling, each task T is subdivided into a potentially infinite sequence of *subtasks*, each with an execution requirement of one quantum, and that the i^{th} subtask of T is denoted T_i, where $i \geq 1$. Each subtask T_i of a synchronous, periodic task is associated with a *pseudo-release* $\mathsf{r}(T_i)$ and a *pseudo-deadline* $\mathsf{d}(T_i)$ defined as in (3.5) and (3.4), respectively, which are repeated here for convenience.

$$\mathsf{r}(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{5.3}$$

$$\mathsf{d}(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \tag{5.4}$$

Each subtask must be scheduled in the interval $[\mathsf{r}(T_i), \mathsf{d}(T_i))$, termed its *window*.

To guide us in the assignment of the jobs of a migratory task to its processors, we define the notion of a *complementary task*.

Definition 5.1: Task T is said to be *complementary* to U iff $wt(U) = 1 - wt(T)$.

Tasks T and U shown in Figure 5.6 are complementary to one another. A partial Pfair schedule for these two tasks on one processor, in which the subtasks of T are always scheduled in the last slots of their windows and those of U in the first slots, is also shown. We call such a schedule a *complementary schedule*. By Lemma 5.1 below such a schedule is always possible for two complementary periodic tasks.

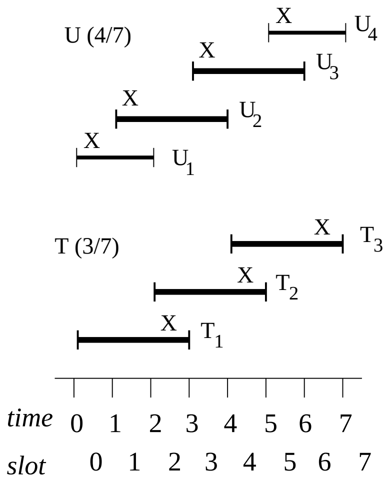


Figure 5.6: A partial complementary Pfair schedule for a pair of complementary tasks, T and U, on one processor. The slot in which a subtask is scheduled is indicated by an "X." Every subtask of U is scheduled in the first slot of its window, while every subtask of T is scheduled in the last slot.

Lemma 5.1 *For any two synchronous, periodic tasks T and U that are complementary, a schedule in which every subtask of T is scheduled in the first slot of its window and every subtask of U in its last slot, or vice versa, is feasible on one processor.*

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

By (5.3), $r(T_i) = t$ implies that $\left\lfloor \frac{i-1}{wt(T)} \right\rfloor = t$ holds. Therefore,

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

$$
\begin{aligned}
&\Rightarrow \quad t - i \le t(1 - wt(T)) - 1 \\
&\Rightarrow \quad \frac{t-i+1}{1-wt(T)} \le t.
\end{aligned} \tag{5.5}
$$

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

$$
\begin{aligned}
& \frac{i-1}{wt(T)} < t+1 \\
\Rightarrow\ & i-1 < wt(T) + t \cdot wt(T) \\
\Rightarrow\ & t-i+1+wt(T) > t(1-wt(T)) \\
\Rightarrow\ & \frac{t-i+1+wt(T)}{1-wt(T)} > t \\
\Rightarrow\ & \frac{t-i+1+wt(T)}{1-wt(T)} + \frac{1-wt(T)}{1-wt(T)} > t + \frac{1-wt(T)}{1-wt(T)} \\
\Rightarrow\ & \frac{t-i+2}{1-wt(T)} > t+1
\end{aligned} \tag{5.6}
$$

By (5.5) and (5.6), it follows that there does not exist an integer j for which $\left\lceil \frac{j}{1-wt(T)} \right\rceil = t+1$ holds. By Definition 5.1, $wt(U) = 1 - wt(T)$. Therefore, by (5.4), it follows that there does not exist a subtask U_j such that $d(U_j) = t+1$ holds. ■

With the above introduction to Pfair scheduling, we are now ready to present the details of distributing the jobs of a migrating task between its processors.

5.1.2.2 Assignment Rules for Jobs of Migrating Tasks

Let τ_i be any migrating periodic task (we later relax the assumption that τ_i is periodic) that is assigned shares $s_{i,j}$ and $s_{i,j+1}$ on processors P_j and P_{j+1}, respectively. (Recall that every migrating task is assigned shares on two consecutive processors by Assign-Tasks.) As explained earlier, $f_{i,j}$ and $f_{i,j+1}$ (given by (5.2)) denote the fraction of the workload (*i.e.*, the total execution requirement) of T that should be executed on P_j and P_{j+1}, respectively, in the long run. By (P1), the total share allocated to τ_i on P_j and P_{j+1} is u_i. Hence, by (5.2), it follows that

$$f_{i,j} + f_{i,j+1} = 1. \tag{5.7}$$

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

Then, by (5.7), it follows that $f_{i,j+1} = \frac{y_i - x_{i,j}}{y_i}$. Therefore, one way of distributing the workload of τ_i between P_j and P_{j+1} that is commensurate with the shares of τ_i on the two processors would be to assign $x_{i,j}$ out of every y_i jobs to P_j and the remaining jobs to P_{j+1}.

Rather than arbitrarily choosing the $x_{i,j}$ jobs to assign to P_j, we borrow from the aforementioned concepts of Pfair scheduling to guide in the distribution of jobs. For illustration, consider a migrating task τ_i with utilization $\frac{3}{8}$ that is assigned shares $s_{i,j} = \frac{1}{5}$ and $s_{i,j+1} = \frac{7}{40}$ on P_j and P_{j+1}, respectively. Hence, $f_{i,j} = \frac{8}{15}$ and $f_{i,j+1} = \frac{7}{15}$ hold. Therefore, one way of distributing τ_i's jobs would be to assign the first eight of jobs $15k+1, \cdots, (15k+15)$, for all $k \geq 0$, to P_j, and the remaining jobs to P_{j+1}. Though such a strategy is reasonable (and perhaps among the best) for the example in Figure 5.5, the distribution may be significantly uneven over short durations for some task systems, such as the present example, and the transient overload that ensues may be quite excessive. As we shall see, a more even distribution of jobs can be obtained by applying Pfair rules.

If we let two fictitious periodic Pfair tasks V and W correspond to processors P_j and P_{j+1}, respectively, let $f_{i,j}$ and $f_{i,j+1}$ denote their weights, and let a quantum span p_i time units, then the following analogy can be made between the jobs of the migrating task τ_i and the subtasks of the fictitious tasks V and W. First, slot s can be associated with job $s+1$ in that slot s represents the interval in which the $(s+1)^{\text{st}}$ job of τ_i, which is released at the beginning of that slot, needs to be scheduled. (Recall that slots are numbered starting from 0.) This is illustrated in Figure 5.7(a), which depicts the layouts of subtasks in the first period of V and W for the example mentioned above, and a complementary schedule for those subtasks on a fictitious processor. Refer to "slots" and "jobs" marked beneath the time line. Continuing with the analogy, subtask V_g represents the g^{th} job assigned to P_j (of the jobs of τ_i); that is, exactly one of the jobs that correspond to slots $\mathsf{r}(V_g), \cdots, \mathsf{d}(V_g) - 1$ should be assigned as the g^{th} job of P_j. Similarly, subtask W_h represents the h^{th} job assigned to P_{j+1}. Finally, if subtask V_g (resp., W_h) is scheduled in slot s (on a fictitious processor), then the $(s+1)^{\text{st}}$ job of τ_i should be assigned to P_j (resp., P_{j+1}). In other words, job $s+1$ is assigned to the processor that corresponds to the Pfair task that is scheduled in slot $s+1$. Referring to Figure 5.7(a), since subtask V_1 is scheduled in slot 0, the first job of τ_i is assigned to P_j. Similarly, since subtasks of V are scheduled in slots 1, 3, 5, 7, 9, 11, and 13, jobs 2, 4, 6, 8, 10, 12, and 14 of τ_i are assigned to P_j, and since subtasks of W are scheduled in slots 2, 4, 6, 8, 10, 12, and 14, jobs 3, 5, 7, 9, 11, 13, and 15 of τ_i are assigned to P_{j+1}.

By Definition 5.1 and (5.7), Pfair tasks V and W are complementary. Therefore, by

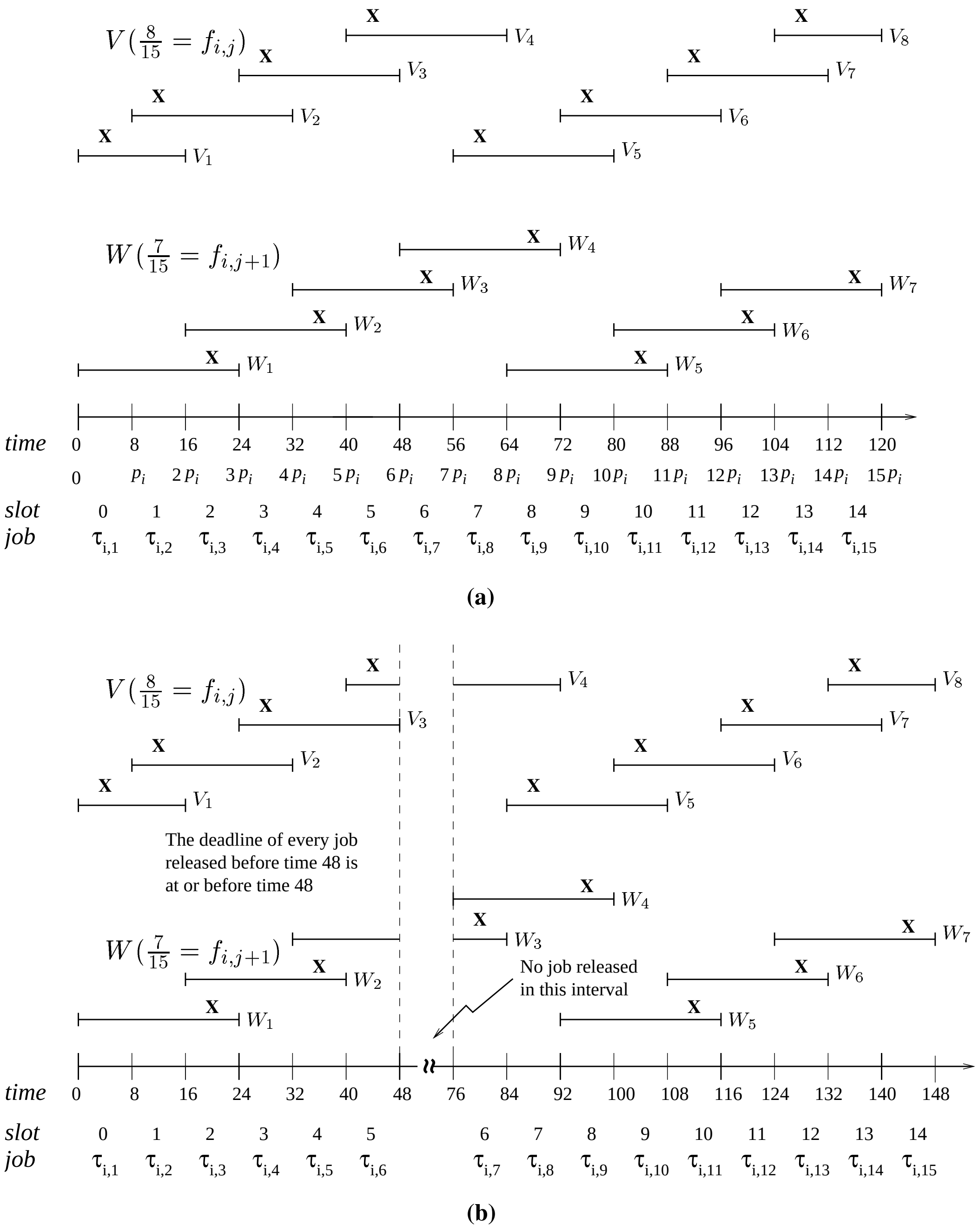


Figure 5.7: Complementary Pfair schedule for tasks V and W with weights $f_{i,j} = 8/15$ and $f_{i,j+1} = 7/15$, respectively, that guides the assignment of jobs of task $\tau_i(3, 8)$ to processors P_i and P_{j+1}. Subtasks in the first period, $[0, 15)$, of V and W are shown. The pattern repeats for every period. Slot k corresponds to job $k+1$ of τ_i. The slot in which a subtask is scheduled is indicated by an "X." **(a)** The jobs of τ_i are released periodically. **(b)** The seventh job of τ_i is delayed by 28 time units.

Lemma 5.1, a complementary schedule for V and W in which the subtasks of V are scheduled in the first slot of their windows and those of W in the last slot of their windows is feasible. Further, because $wt(V) + wt(W) = 1$, some subtask is scheduled in each slot. Hence, the following holds.

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

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

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

More generally, we can use the formula for the release time of a subtask given by (5.3) for job assignments. Let job_i denote the total number of jobs released by task τ_i prior to time t and let $job_{i,j}$ denote the total number of jobs of τ_i that have been assigned to P_j before t. Let $p_{i,\ell}$ denote the processor to which job ℓ of task τ_i is assigned. Then, the processor to which job $job_i + 1$, released at or after time t, is assigned is determined as follows.

$$p_{i,job_i+1} = \begin{cases} j, & \text{if } job_i = \left\lfloor \dfrac{job_{i,j}}{f_{i,j}} \right\rfloor \\ j+1, & \text{otherwise} \end{cases} \tag{5.8}$$

As before, let $f_{i,j}$ and $f_{i,j+1}$ be the weights of two fictitious Pfair tasks V and W, respectively. Then, by (5.3), $t_r = \left\lfloor \dfrac{job_{i,j}}{f_{i,j}} \right\rfloor$ denotes the release time of subtask $V_{job_{i,j}+1}$ of task V. Thus, (5.8) assigns to P_j the job that corresponds to the first slot in the window of subtask V_g as the g^{th} job of τ_i on P_j, for all g. (Recall that the index of the job of the migrating periodic task τ_i that is released in slot t_r is given by $t_r + 1$.) Because the sum of the weights of the two tasks is one, Lemma 5.1 implies that if t_r is not the release time of any subtask of V, then $t_r + 1$ is the deadline of some subtask of W. Thus, (5.8) ensures that the job that corresponds to the last slot in the window of subtask W_h is assigned as the h^{th} job of τ_i on P_{j+1}, for all h.

Thus far in our discussion, to simplify the presentation, we have assumed that the job releases of task τ_i are periodic. However, note that the job assignment given by (5.8) is independent of "real" time and is based on job numbers only. Hence, assigning jobs using (5.8) should be sufficient to ensure (A2) even when τ_i is sporadic. This is illustrated in Figure 5.7. Here, we assume that τ_i is a sporadic task, whose seventh job release is delayed, by 28 time units, to time 76 from time 48. As far as τ_i is concerned, the interval $[48, 76)$ is "frozen" and the job assignment resumes at time 76. As indicated in the figure, in any such interval in which activity is suspended for a migrating task τ_i, no jobs of τ_i are released. Furthermore, the deadlines of all jobs of τ_i released before the frozen interval fall at or before the beginning of the interval.

We next prove a property that bounds from above the number of jobs of a migrating task assigned to each of its processors by the job assignment rule given by (5.8).

Lemma 5.2 *Let τ_i be a migrating task that is assigned to processors P_j and P_{j+1}. The number of jobs out of any consecutive $\ell \geq 0$ jobs of τ_i that are assigned to P_j and P_{j+1} is at most $\lceil \ell \cdot f_{i,j} \rceil$ and $\lceil \ell \cdot f_{i,j+1} \rceil$, respectively.*

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

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

(J) holds trivially when $\ell_0 = 0$. Therefore, assume $\ell_0 \geq 1$. Let q denote the total number of jobs of the first ℓ_0 jobs of τ_i that are assigned to P_j. (By (5.8), the first job of τ_i is assigned to P_j, hence, $q \geq 1$ holds.) Then, there exists an $\ell' \leq \ell_0$ such that job ℓ' of τ_i is the q^{th} job of τ_i assigned to P_j. Therefore, by (5.8),

$$\ell' - 1 = \left\lfloor \frac{q-1}{f_{i,j}} \right\rfloor \tag{5.9}$$

holds. (Note that job_i of (5.8) denotes the number of jobs of τ_i that have already been distributed, and hence, is equal to $\ell - 1$ here. Similarly, $job_{i,j}$ denotes the number of jobs already assigned to to P_j, and so is equal to $q - 1$.) ℓ, ℓ', and q denote job numbers or counts, and hence are all non-negative integers. By (5.9), we have

$$\frac{q-1}{f_{i,j}} \geq \ell' - 1 \quad \Rightarrow \quad q - 1 \geq (\ell' - 1) \cdot f_{i,j} \quad \Rightarrow \quad q > \ell' \cdot f_{i,j} \qquad \text{(because } f_{i,j} < 1\text{)}, \tag{5.10}$$

and

$$\frac{q-1}{f_{i,j}} < \ell' \quad \Rightarrow \quad q - 1 < \ell' \cdot f_{i,j} \quad \Rightarrow \quad q < \ell' \cdot f_{i,j} + 1. \tag{5.11}$$

Because q is an integer, by (5.10) and (5.11), we have

$$q = \lceil \ell' \cdot f_{i,j} \rceil. \tag{5.12}$$

If $\ell' = \ell_0$ holds, then (J) follows from (5.12) and our definition of q. On the other hand, to show that (J) holds when $\ell' < \ell_0$, we must show that $q = \lceil \hat{\ell} \cdot f_{i,j} \rceil$ holds for all $\hat{\ell}$, where $\ell' < \hat{\ell} \leq \ell_0$. (Note that $\hat{\ell}$ is an integer.) By the definitions of q, ℓ', and ℓ_0, q of the first ℓ' jobs of τ_i are assigned to P_j, and none of the jobs $\ell' + 1$ through ℓ_0 is assigned to P_j. Therefore, by (5.8), it follows that $\hat{\ell} - 1 < \left\lfloor \frac{q}{f_{i,j}} \right\rfloor$ holds for all $\hat{\ell}$, where $\ell' < \hat{\ell} \leq \ell_0$. (As before, because $\hat{\ell}$ is the index of the next job of τ_i to be distributed, job_i of (5.8) equals $\hat{\ell} - 1$. However, since the number of jobs already assigned to P_j is q, $job_{i,j} = q$.) Thus, we have the following, for all $\hat{\ell}$, where $\ell' < \hat{\ell} \leq \ell_0$.

$$\left\lfloor \frac{q}{f_{i,j}} \right\rfloor > \hat{\ell} - 1 \Rightarrow \left\lfloor \frac{q}{f_{i,j}} \right\rfloor \geq \hat{\ell} \Rightarrow \frac{q}{f_{i,j}} \geq \hat{\ell} \Rightarrow q \geq \hat{\ell} \cdot f_{i,j} \Rightarrow q \geq \lceil \hat{\ell} \cdot f_{i,j} \rceil \quad \text{(because } q \text{ is an integer)} \tag{5.13}$$

By (5.12) and because $\hat{\ell} > \ell'$ holds, (5.13) implies that $\lceil \hat{\ell} \cdot f_{i,j} \rceil = \lceil \ell' \cdot f_{i,j} \rceil = q$.

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

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

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

Finally, we are left with proving the lemma for P_{j+1}. By the job assignment rule in (5.8), every job of τ_i is assigned to exactly one of P_j and P_{j+1}. Therefore (J) implies that exactly $\ell_0 - \lceil \ell_0 \cdot f_{i,j} \rceil$ of the first ℓ_0 jobs of τ_i are assigned to P_{j+1}. Hence, the number of jobs belonging to $\mathcal{J}$ that are assigned to P_{j+1} is given by

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

$$= \quad \ell \cdot f_{i,j+1} + 1 \qquad\qquad \text{(by (5.7))}.$$

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

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

5.1.3 Tardiness Bound for EDF-fm

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

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

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

$$t_d \stackrel{\text{def}}{=} \text{absolute deadline of job } \tau_{q,\ell} \tag{5.14}$$

$$t_c \stackrel{\text{def}}{=} t_d + \Delta \tag{5.15}$$

$$t_0 \stackrel{\text{def}}{=} \begin{array}{l}\text{latest instance before } t_c \text{ such that at } t_0 - \epsilon,\ P_k \text{ either is idle} \\ \text{or is executing a job of a fixed task with a deadline later} \\ \text{than } t_d\end{array} \tag{5.16}$$

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

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

By (5.16) and because migrating tasks have a higher priority than fixed tasks under EDF-fm, jobs of τ_i and τ_j that are released before t_0 and are assigned to P_k complete executing at or before t_0. Thus, every job of τ_i or τ_j that executes in $[t_0, t_c)$ on P_k is released in $[t_0, t_c)$. Also, every job released in $[t_0, t_c)$ and assigned to P_k places a demand for execution in $[t_0, t_c)$. The number of jobs of τ_i that are released in $[t_0, t_c)$ is at most $\left\lceil \frac{t_c - t_0}{p_i} \right\rceil$. By Lemma 5.2, at most $\left\lceil f_{i,k} \left\lceil \frac{t_c - t_0}{p_i} \right\rceil \right\rceil \leq f_{i,k} \left(\frac{t_c - t_0}{p_i} + 1 \right) + 1$ of all the jobs of τ_i released in $[t_0, t_c)$ are assigned to P_k. Similarly, the number of jobs of τ_j that are assigned to P_k of all jobs of τ_i released in $[t_0, t_c)$ is at most $f_{j,k} \left(\frac{t_c - t_0}{p_j} + 1 \right) + 1$. Each job of τ_i executes for at most e_i time units and that of τ_j for e_j time units. Therefore,

$$\begin{aligned}
demand(\tau_k^m, t_0, t_c) &\leq \left(f_{i,k} \left(\frac{t_c - t_0}{p_i} + 1 \right) + 1 \right) \cdot e_i + \left(f_{j,k} \left(\frac{t_c - t_0}{p_j} + 1 \right) + 1 \right) \cdot e_j \\
&= s_{i,k}(t_c - t_0) + e_i(f_{i,k} + 1) + s_{j,k}(t_c - t_0) + e_j(f_{j,k} + 1) \qquad (5.17) \\
&\qquad \text{(by (5.2) and simplification).}
\end{aligned}$$

By (5.14)–(5.16), and our assumption that the tardiness of $\tau_{q,\ell}$ exceeds Δ, any job of a fixed task that executes on P_k in $[t_0, t_c)$ is released at or after t_0 and has a deadline at or before t_d. The number of such jobs of a fixed task τ_f is at most $\left\lfloor \frac{t_d - t_0}{p_f} \right\rfloor$. Therefore,

$$\begin{aligned}
demand(\tau_k^f, t_0, t_c) &\leq \sum_{\tau_f \in \tau_k^f} \left\lfloor \frac{t_d - t_0}{p_f} \right\rfloor \cdot e_f \\
&\leq (t_d - t_0) \sum_{\tau_f \in \tau_k^f} \frac{e_f}{p_f} \\
&\leq (t_d - t_0)(\rho_k - s_{i,k} - s_{j,k}) \qquad \text{(by (P3)).} \qquad (5.18)
\end{aligned}$$

By (5.17) and (5.18), we have the following.

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

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

$$\begin{aligned}
t_c - t_0 &< (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) + \rho_k(t_d - t_0) \\
&= (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) + (t_d - t_0) \\
&\quad -(1 - \rho_k)(t_d - t_0) \\
\Rightarrow\ t_c - t_d &< (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) - (1 - \rho_k)(t_d - t_0) \\
&\leq (s_{i,k} + s_{j,k})(t_c - t_d) + e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) - p_q(1 - \rho_k) \\
&\qquad \text{($\tau_{q,\ell}$ is released at or after t_0 and has a deadline at t_d, hence $t_d - t_0 \geq p_q$)} \\
\Rightarrow\ t_c - t_d &< \frac{e_i(f_{i,k} + 1) + e_j(f_{j,k} + 1) - p_q(1 - \rho_k)}{1 - s_{i,k} - s_{j,k}} = \Delta. \qquad (5.19)
\end{aligned}$$

The above contradicts (5.15), and hence our assumption that the tardiness of $\tau_{q,\ell}$ exceeds Δ is incorrect. Therefore, (L) follows.

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

$$\frac{e_i(f_{i,k} + 1) - p_q(1 - \rho_k)}{1 - s_{i,k}}. \qquad (5.20)$$

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

Theorem 5.1 *On M processors, Algorithm* EDF-fm *ensures a tardiness of at most*

$$\frac{e_{m_{k,1}}(f_{m_{k,1},k} + 1) + e_{m_{k,2}}(f_{m_{k,2},k} + 1) - p_q(1 - \rho_k)}{1 - s_{m_{k,1},k} - s_{m_{k,2},k}} \qquad (5.21)$$

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

Because (5.21) can be computed in constant time, the overall time complexity of computing a tardiness bound for τ is $\mathcal{O}(N)$. (5.21) increases as the execution costs and shares of the migrating tasks assigned to P_k increase, and could be high if the share of each migrating task is close to 1/2. However, because all tasks are light, in practice the sum of the shares of the migrating tasks assigned to a processor can be expected to be less than 1/2. Theorem 5.1 also suggests that the tardiness that results in practice could be reduced by choosing the set of migrating tasks carefully. Tardiness can also be reduced by distributing smaller pieces of work of migrating tasks than entire jobs. Some such techniques and heuristics are discussed in the next section.

5.2 Tardiness Reduction Techniques for EDF-fm

The problem of assigning tasks to processors such that the tardiness bound given by (5.21) is minimized is a combinatorial optimization problem with exponential time complexity. Hence, in this section, we propose methods and heuristics that can lower tardiness. We consider the technique of *period transformation* [99] as a way of distributing the execution of jobs of migrating tasks more evenly over their periods in order to reduce their adverse impact on fixed tasks. We also propose task assignment heuristics that can reduce the fraction of a processor's capacity consumed by migrating tasks. Finally, we show how to compute more accurate bounds than that given by (5.21) at the expense of more complex computations.

5.2.1 Job Slicing

The tardiness bound of EDF-fm given by Theorem 5.1 is in multiples of the execution costs of migrating tasks. This is a direct consequence of statically prioritizing migrating tasks over fixed tasks and the overload (in terms of the number of jobs) that a migrating task may place on a processor over short intervals. The deleterious effect of this approach on jobs of fixed tasks can be mitigated by "slicing" each job of a migrating task into *sub-jobs* that have lower execution costs, assigning appropriate deadlines to the sub-jobs, and distributing and scheduling sub-jobs in the place of whole jobs. For example, every job of a task with an execution cost of 4 time units and relative deadline of 10 time units can be sliced into two sub-jobs with execution cost and relative deadline of 2 and 5, respectively, per sub-job, or four sub-jobs with an execution cost of 1 and relative deadline of 2.5, per sub-job. Such a job-slicing approach, termed *period transformation*, was proposed by Sha and Goodman [99] in the context of RM scheduling on uniprocessors. Their purpose was to boost the priority of tasks that have larger periods, but are more important than some other tasks with shorter periods, and thus ensure that the more important tasks do not miss deadlines under overloads. However, with the job-slicing approach under EDF-fm, it may be necessary to migrate a job between its processors, and EDF-fm loses the property that a task that migrates does so only across job boundaries. Thus, this approach presents a trade-off between tardiness and migration overhead.

5.2.2 Task-Assignment Heuristics

Another way of lowering the actual tardiness observed in practice would be to lower the total share $s_{m_{k,1},k} + s_{m_{k,2},k}$ assigned to the migrating tasks on any processor P_k. In the task assignment algorithm ASSIGN-TASKS of Figure 5.1, if a low-utilization task is ordered between two high-utilization tasks, then it is possible that $s_{m_{k,1},k} + s_{m_{k,2},k}$ is arbitrarily close to one. For example, consider tasks τ_{i-1}, τ_i, and τ_{i+1} with utilizations $\frac{1-\epsilon}{2}$, 2ϵ, and $\frac{1-\epsilon}{2}$, respectively, and a task assignment wherein τ_{i-1} and τ_{i+1} are the migrating tasks of P_k with shares of $\frac{1-2\epsilon}{2}$ each, and τ_i is the only fixed task on P_k. Such an assignment, which can delay τ_i excessively if the periods of τ_{i-1} and τ_{i+1} are large, can be easily avoided by ordering tasks by (monotonically) decreasing utilization prior to the assignment phase. Note that with tasks ordered by decreasing utilization, of all the tasks not yet assigned to processors, the one with the highest utilization is always chosen as the next migrating task. Hence, we call this assignment scheme *highest utilization first*, or HUF. An alternative *lowest utilization first*, or LUF, scheme can be defined that assigns fixed tasks in the order of (monotonically) decreasing utilization, but chooses the task with the lowest utilization of all the unassigned tasks as the next migrating task. Such an assignment can be accomplished using the following procedure when a migrating task needs to be chosen: traverse the unassigned task array in reverse order starting from the task with the lowest utilization and choose the first task whose utilization is at least the capacity available in the current processor. In general, this scheme can be expected to lower the shares of migrating tasks. However, because the unassigned tasks have to be scanned each time a migrating task is chosen, the time complexity of this scheme increases to $\mathcal{O}(NM)$ (from $\mathcal{O}(N)$). This complexity can be reduced to $\mathcal{O}(N + M \log N)$ by adopting a binary-search strategy.

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

5.2.3 Including Heavy Tasks

The primary reason for restricting all tasks to be light is to prevent the total utilization $u_i + u_j$ of the two migrating tasks τ_i and τ_j assigned to a processor from exceeding one. (As already noted, ensuring that migrating tasks do not miss their deadlines may not be possible otherwise.) However, if the number of heavy tasks is small in comparison to the number of light tasks, then it may be possible to avoid an undesirable assignment as described. In the simulation experiments discussed in Section 5.3, with no restrictions on per-task utilizations, the LUF approach could successfully assign approximately 78% of one million randomly-generated task sets on 4 processors. The success ratio dropped to approximately one-half when the number of processors increased to 16.

5.2.4 Processors with One Migrating Task

If the number of migrating tasks assigned to a processor P_k is one, then the commencement of the execution of a job $\tau_{i,j}$ of the only migrating task τ_i of P_k can be postponed to time $d(\tau_{i,j}) - e_i$, where $d(\tau_{i,j})$ is the absolute deadline of job $\tau_{i,j}$ (instead of beginning its execution immediately upon its arrival). This would reduce the maximum tardiness of the fixed tasks on P_k to $(e_i - p_q(1 - \rho_k))/(1 - s_{i,k})$ (from the value given by (5.20)). The reasoning is as follows. From the analysis in Section 5.1.3, tardiness of fixed tasks is bounded when the migrating task is not deferred, and hence, by the same analysis, is guaranteed to be bounded with deferred execution. This in turn implies that an arbitrary job of any fixed task completes execution. Taking t_c as the completion time of job $\tau_{q,\ell}$ when all the jobs execute for their worst-case execution times, where $\tau_{q,\ell}$ is as defined in Section 5.1.3, no job of τ_i with deadline later than t_c executes before t_c. (This is because, under deferred execution, each job of τ_i completes executing at its deadline. Hence, if $\tau_{q,\ell}$ completes execution at t_c, then neither is t_c the deadline of any job of τ_i nor does the first job with deadline after t_c commence execution by t_c.) Therefore, the number of jobs of τ_i released in the interval $[t_0, t_c)$ that can impact $\tau_{q,\ell}$ is at most $\left\lfloor \frac{t_c - t_0}{p_i} \right\rfloor \leq \frac{t_c - t_0}{p_i}$. This is one fewer job than that possible in the absence of deferred execution, which helps lower the tardiness bound derived by $\frac{e_i \cdot f_{i,k}}{1 - s_{i,k}}$. (Because tasks are independent and are preemptable, tardiness is guaranteed to not increase when one or more jobs execute for less than their worst-case execution times.) This technique is likely to be particularly effective on two-processor systems, where each processor would be assigned at most one migrating task only under EDF-fm, and on three-processor systems, where at most one processor would be assigned two migrating tasks.

5.2.5 Computing More Accurate Tardiness Bounds

Thus far in this section, we have discussed some techniques that can be used to lower the tardiness observed in practice and the bound computed using (5.21). We now describe how a more accurate tardiness bound can be computed for a given task assignment.

One major source of pessimism in the bound is the approximation of ceiling and floor operations during the analysis. This could be eliminated at the expense of more complex computations, wherein a bound is computed by iteratively computing a worst-case *response time*[3] for each task. The approach is similar to the time-demand [77] and the generalized time-demand analyses [75] used in conjunction with static-priority algorithms, and the response-time analysis developed for systems scheduled under EDF on uniprocessors [102].

Before continuing further, some definitions are in order. An interval $[t_1, t_2)$ is said to be *busy* for Processor P_k if the following hold. (In the description of a busy interval that follows, by P_k's tasks, we refer to both its fixed and migrating tasks, and by jobs of P_k's migrating tasks, we refer to jobs that are assigned to P_k.) **(i)** No job of any of P_k's tasks that is released before t_1 is pending at t_1; **(ii)** one or more jobs of P_k's tasks are released at t_1; and **(iii)** t_2 is the earliest time after t_1 such that no job released before t_2 is pending. Note that (i)–(iii) imply that P_k is continuously busy in $[t_1, t_2)$. A busy interval $[t_1, t_2)$ is said to be *in-phase for a fixed task* τ_i if a job of τ_i is released at t_1 and *in-phase for a migrating task* τ_i if a job of τ_i that begins a worst-case assignment sequence for P_k is released at t. By Lemma 5.2, at most $\lceil \ell \cdot f_{i,k} \rceil$ of any ℓ consecutive jobs of a migrating task τ_i are assigned to P_k. Therefore, $[t_1, t_2)$ is in-phase for τ_i if some job of τ_i is released at t_1, and if $\lceil \ell_t \cdot f_{i,k} \rceil$ of the jobs of τ_i are assigned to P_k in the interval $[t_1, t)$ for all $t_1 \le t < t_2$, where ℓ_t denotes the number of jobs of τ_i released in that interval. A busy interval is said to be *tight* if all tasks release jobs as early as permissible after the release of their first jobs in the interval.

In [102], the following has been shown for a task system scheduled under EDF on a uniprocessor: The largest response time of any job of a task τ_i (all tasks are fixed on a uniprocessor) released in a tight, busy interval that is in-phase for every task except perhaps τ_i is not lower than that of any job of τ_i released in any busy interval. Under EDF-fm, the same can be

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

shown to hold for every fixed task. The reasoning is transformation-based and is as follows: By definition, no job is pending at the beginning of a busy interval; hence, transforming a busy interval that is not in-phase for a task τ_j (which is either fixed or migrating), by shifting left its jobs released in the interval, such that τ_j is in-phase, cannot decrease the demand due to τ_j in the busy interval that can compete with τ_i's jobs. Similarly, the demand due to τ_j cannot decrease if its job releases are made tight, *i.e.*, if τ_j's jobs are released as early as permissible.

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

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

Computing the longest busy interval length. A tight, busy interval that is in-phase for every task, including τ_i, is at least as long as any busy interval that is not in-phase for τ_i. Therefore, we will upper bound the lengths of busy intervals we are interested in by that of one that is tight and in-phase for all tasks. For brevity, we will refer to such an interval as simply a busy interval. Without loss of generality, we assume that the longest busy interval that we are considering starts at time zero. Letting B_k denote the length of a longest busy interval of P_k, B_k can be computed iteratively as follows. As in Section 5.1.3, τ_k^m and τ_k^f refer to the sets of fixed and migrating tasks, respectively, assigned to P_k. B_k^0 denotes the initial value of B_k and is given by the following (since each task has a job released at the start of the interval that needs to complete execution).

$$B_k^0 = \sum_{\tau_h \in \tau_k^m} e_h + \sum_{\tau_h \in \tau_k^f} e_h \tag{5.22}$$

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

$$B_k^{i+1} = \sum_{\tau_h \in \tau_k^m} \left\lceil \left\lceil \frac{B_k^i}{p_h} \right\rceil \cdot f_{h,k} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau_k^f} \left\lceil \frac{B_k^i}{p_h} \right\rceil \cdot e_h. \tag{5.23}$$

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

$$B_k = (B_k^i \mid B_k^i = B_k^{i+1} \ \wedge \ 0 \leq i \leq \min_{j \geq 0}\{j \mid B_k^j = B_k^{j+1}\})$$

We next show that termination is guaranteed. For simplicity, we assume that the execution costs and periods of all tasks are integers. Let $s_{h,k} = \frac{x_{h,k}}{y_{h,k}}$, where $x_{h,k}$ and $y_{h,k}$ are positive integers that are relatively prime. Let Δ denote the least common multiple (lcm) of the periods of all fixed and migrating tasks and the product $e_h \cdot y_{h,k}$ for each migrating task τ_h. Then, for all $B_k^i \leq \Delta$, by (5.23) and (5.2), $B_k^{i+1} \leq \sum_{\tau_h \in \tau_k^m} \left\lceil \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot \frac{s_{h,k} \cdot p_h}{e_h} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau_k^f} \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot e_h = \sum_{\tau_h \in \tau_k^m} \left\lceil \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot \frac{x_{h,k} \cdot p_h}{y_{h,k} \cdot e_h} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau_k^f} \left\lceil \frac{\Delta}{p_h} \right\rceil \cdot e_h$. Since Δ is as defined, p_h for each fixed and migrating task, and $e_h \cdot y_{h,k}$ for each migrating task divide Δ evenly, and hence, $B_k^{i+1} \leq \sum_{\tau_h \in \tau_k^m} \frac{\Delta}{p_h} \cdot \frac{x_{h,k} \cdot p_h}{y_{h,k} \cdot e_h} \cdot e_h + \sum_{\tau_h \in \tau_k^f} \frac{\Delta}{p_h} \cdot e_h = \sum_{\tau_h \in \tau_k^m} \Delta \cdot s_{h,k} + \sum_{\tau_h \in \tau_k^f} \frac{\Delta}{p_h} \cdot e_h$. Because the sum of the shares of the migrating tasks and the utilizations of the fixed tasks assigned to each processor is at most one, the right-hand side of the above inequality is at most Δ. Thus, the computation converges at least when $B_k^i = \Delta$, and hence, B_k is at most Δ.

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

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

$$C^0_{q,\ell,\phi_q} = \min(B_k - e_q, (\ell - 1) \cdot p_q + \phi_q) + e_q. \tag{5.24}$$

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

$$C^{i+1}_{q,\ell,\phi_q} = \sum_{\tau_h \in \tau^m_k} \left\lceil \left\lceil \frac{C^i_{q,\ell,\phi_q}}{p_h} \right\rceil \cdot f_{h,k} \right\rceil \cdot e_h + \sum_{\tau_h \in \tau^f_k} \min\left(\left\lceil \frac{C^i_{q,\ell,\phi_q}}{p_h} \right\rceil, \left\lfloor \frac{\ell \cdot p_q + \phi_q}{p_h} \right\rfloor \right) \cdot e_h \tag{5.25}$$

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

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

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

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

Numerical example. Let us consider computing tardiness bounds of fixed tasks assigned to P_1 in the task system in Example 5.1. In this example, $\tau_1^f = \{\tau_1(5, 20), \tau_2(3, 10)\}$ and $\tau_1^m = \{\tau_3(1, 2)\}$. Further, $s_{3,1} = \frac{9}{20}$, hence, $x_{3,1} = 9$ and $y_{3,1} = 20$. Therefore, $e_3 \cdot y_{3,1} = 1 \cdot 20 = 20$. $\mathrm{lcm}(p_1, p_2, p_3, e_3 \cdot y_{3,1}) = \mathrm{lcm}(20, 10, 2, 20) = 20$. Also, $f_{3,1} = s_{3,1} \cdot \frac{p_3}{e_3} = \frac{9}{20} \cdot \frac{2}{1} = \frac{9}{10}$.

We will first compute the length, B_1, of a longest possible busy interval, on P_1. Using (5.22), $B_1^0 = e_3 + e_1 + e_2 = 9$. By (5.23),

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

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

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

We next consider τ_2, for which, ϕ_2 is in the range $[0, 9]$. The number of jobs of τ_2 for which worst-case response times need to be determined is given by $J - 1 = \left\lceil \frac{B_1 - \phi_2}{p_2} \right\rceil - 1 = \left\lceil \frac{20-\phi_2}{10} \right\rceil - 1 = 1$, for all $0 \le \phi_2 \le 9$. We compute the response time for the first job (*i.e.*, $\ell = 1$) of τ_2, which is $\tau_{2,1}$, when $\phi_2 = 0$. By (5.24), $C_{2,1,0}^0 = \min(20 - 3, 0) + 3 = 3$. By (5.25),

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

Thus, convergence is reached after four iterations and the worst-case response time of $\tau_{2,1} = 6$. Since $d(\tau_{2,1}) = 10$, $\tau_{2,1}$'s tardiness is zero. It can similarly be verified that tardiness for $\tau_{2,1}$ is zero for every ϕ_2 in $[1, 9]$. Hence, tardiness bounds for both τ_1 and τ_2 are zero. On the other hand, tardiness bounds computed for τ_1 and τ_2 using the formula in Section 5.2.4 are 9.01 and 5.45, respectively.

5.3 Simulation-Based Evaluation

In this section, we describe the results of four sets of simulation experiments conducted using randomly-generated task sets to evaluate EDF-fm and the heuristics described in Section 5.2.

The experiments in the first set evaluate the various task assignment heuristics for $M = 4$ and $M = 8$ (where M is the number of processors), and $u_{\max} = 0.25$ and $u_{\max} = 0.5$ (where $u_{\max}$ is the maximum utilization of any task in a task set). For each M and $u_{\max}$, 10^6 task sets were generated. Each task set τ was generated as follows: New tasks were added to τ as long as the total utilization of τ was less than M. For each new task τ_i, first, its period p_i was generated as a uniform random number in the range $[1.0, 100.0]$; then, its execution cost was chosen randomly in the range $[u_{max}, u_{\max} \cdot p_i]$. The last task was generated such that the total utilization of τ exactly equaled M. The generated task sets were classified by maximum and average execution costs (denoted $e_{\max}$ and e_{avg}). The tardiness bound given by (5.21) was computed for each task set under a random task assignment and also under heuristics HUF, LUF, and LEF. The average value of the tardiness bound for task sets in each group under each classification and heuristic was then computed. The results for the groups classified by $e_{\max}$ and e_{avg} for $M = 4$ and $u_{\max} = 0.5$ are shown in insets (a) and (b), respectively, of Figure 5.8. Insets (c) and (d) contain the results under the same classifications for the same M but for $u_{\max} = 0.25$. Results for $M = 8$ are shown in Figure 5.9. (99% confidence intervals were also computed but are omitted due to scale.)

Results for task sets grouped by $u_{\max}$ and u_{avg} are shown in Figure 5.10 for $M = 4$ and $M = 8$. Data for these results also come from 10^6 task sets. However, for this subset of experiments, tasks were generated by uniformly choosing an execution cost in the range $[1.0, 20.0]$ and a utilization in the range $[u_{\min}, u_{\max}]$; the pair $(u_{\min}, u_{\max})$ for each task set was uniformly chosen from those in the set $\{(0.0, 0.2), (0.0, 0.4), (0.1, 0.5), (0.3, 0.5)\}$. This strategy was used so that a sufficient number of task sets fall under each u_{avg} group.

From the plots, we first observe that there is only a slight increase in the tardiness bounds

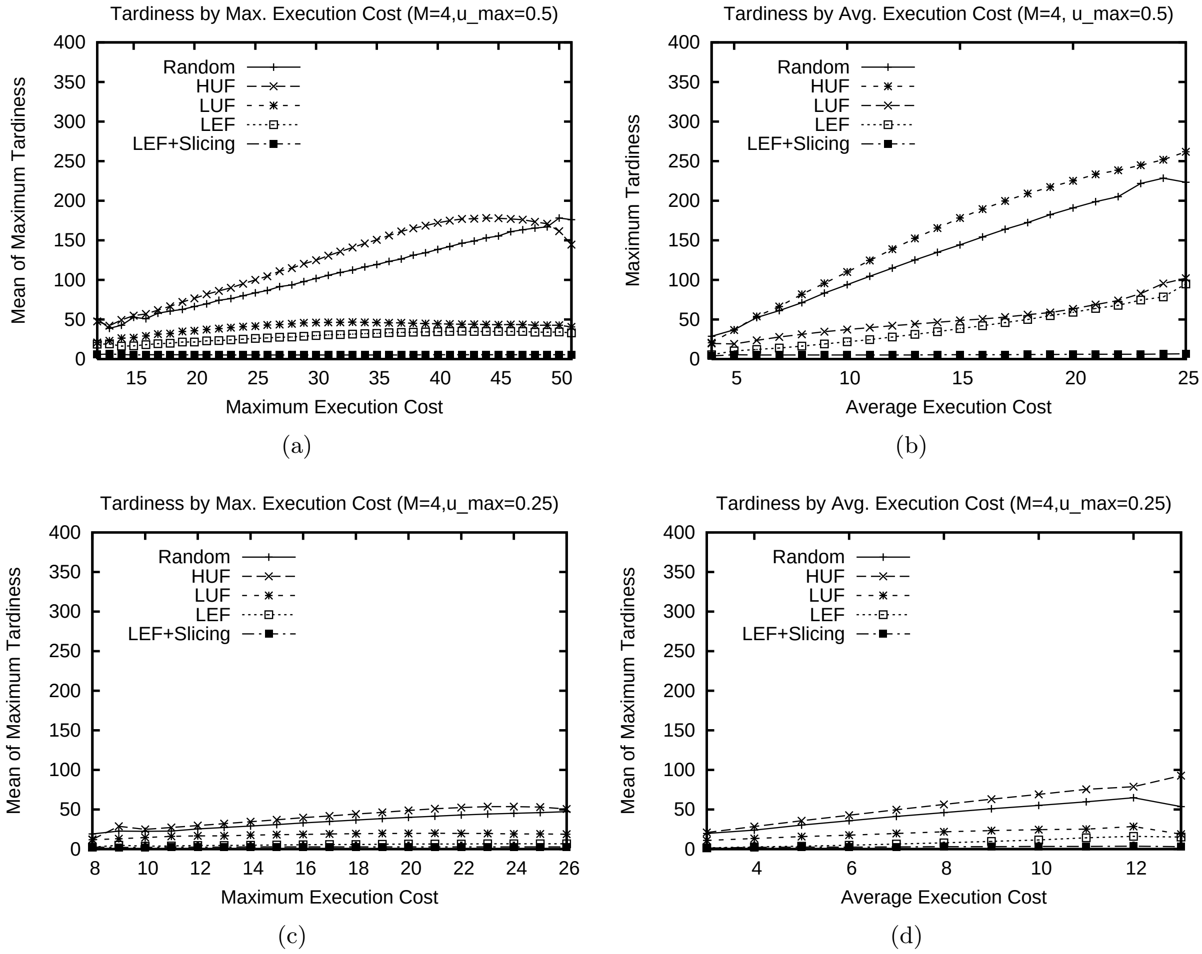


Figure 5.8: Tardiness bounds under different task assignment heuristics for $M = 4$ and $u_{\max} = 0.5$ by **(a)** $e_{\max}$ and **(b)** e_{avg}, and for $M = 4$ and $u_{\max} = 0.25$ by **(c)** $e_{\max}$ and **(d)** e_{avg}.

as the number of processors is increased from four to eight. This is because the tardiness bound given by (5.21) is independent of M. However, the maximum of the tardiness bounds computed for all the tasks can be expected to increase as the number of processors, and hence, the number of tasks increase. The increase, however, seems to be negligible.

Coming to the comparison of the different heuristics, the plots show that LEF guarantees the minimum tardiness of the four task-assignment approaches. LUF is the next best with the difference between LEF and LUF being wider on $M = 8$ processors than on $M = 4$ processors. Another interesting observation is that HUF performs worse than even Random most of the time. Under LEF, tardiness is quite low (approximately 8 time units mostly) for $u_{\max} = 0.25$ (insets (c) and (d) of Figures 5.8 and 5.9 and insets (a) and (c) of Figure 5.10), which

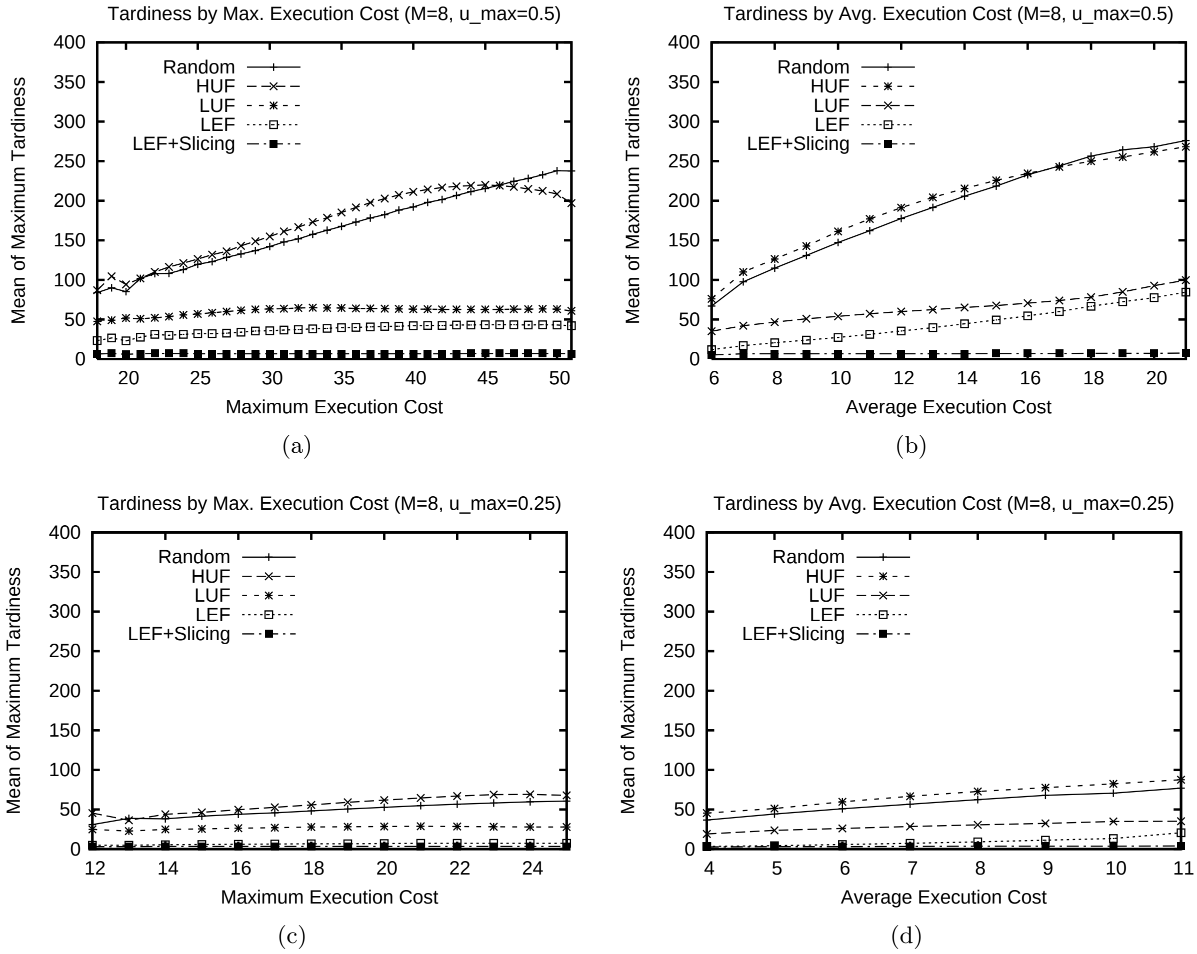


Figure 5.9: Tardiness bounds under different task assignment heuristics for $M = 8$ and $u_{\max} = 0.5$ by **(a)** $e_{\max}$ and **(b)** e_{avg}, and for $M = 8$ and $u_{\max} = 0.25$ by **(c)** $e_{\max}$ and **(d)** e_{avg}.

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

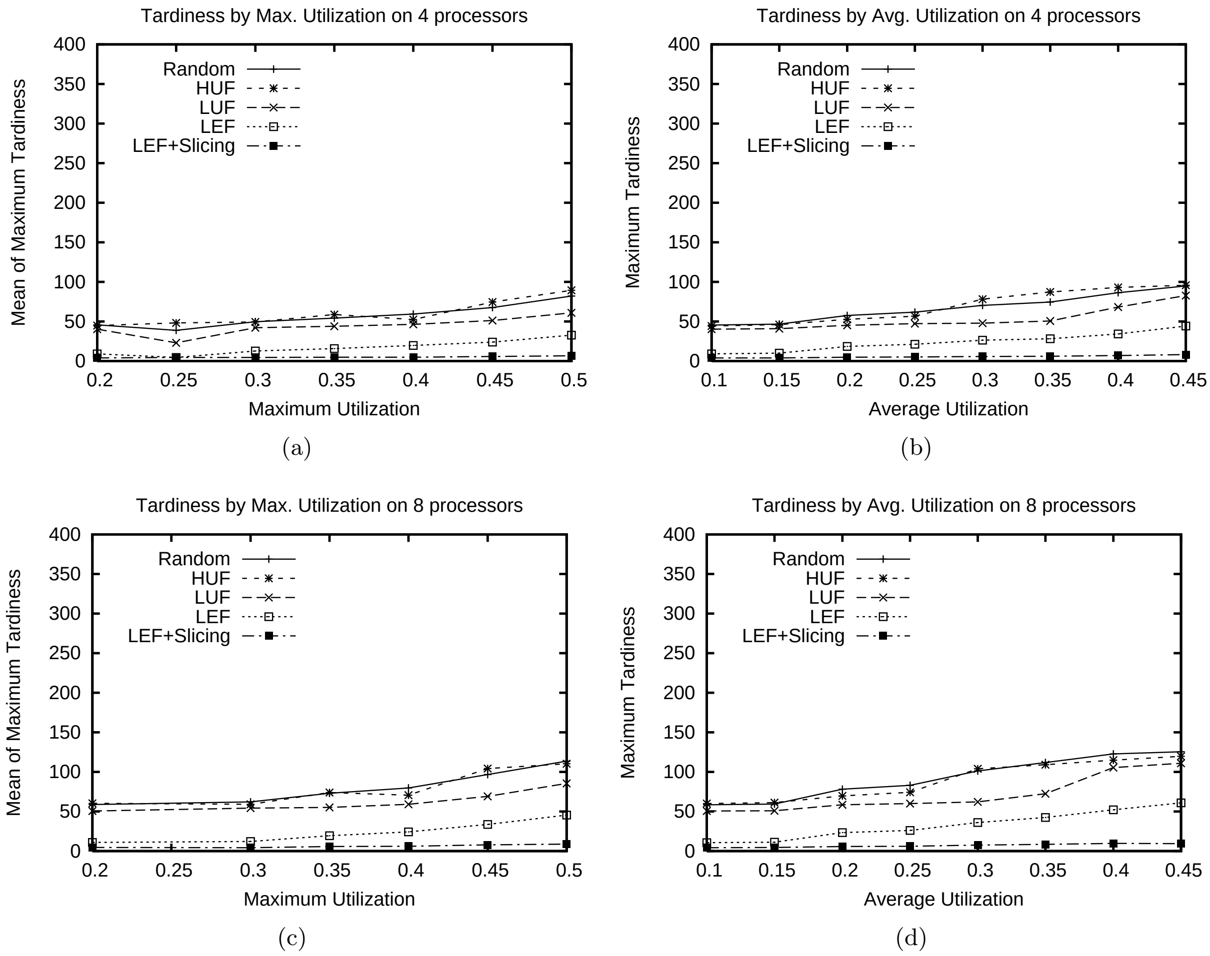


Figure 5.10: Tardiness bounds under different task assignment heuristics for **(a)** $M = 4$ and $u_{\max}$ **(b)** $M = 4$ and u_{avg} **(c)** $M = 8$ and $u_{\max}$ **(d)** $M = 8$ and u_{avg}. In all the graphs, $e_{\max} = 20$ and $e_{\text{avg}} = 10$.

units under LEF with job slicing.) In our experiments, on average, a seven-fold decrease in tardiness was observed with job slicing with a granularity of one to two time units per sub-job. However, a commensurate increase in the number of migrations is also inevitable.

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

The second set of experiments evaluates the different heuristics in their ability to successfully assign task sets that contain heavy tasks also. Task sets were generated using the same procedure as that described for the first set of experiments above, except that $u_{\max}$ was varied between 0.6 and 1.0 in steps of 0.1. All of the four approaches could assign 100% of the task sets generated for $M = 2$, as expected. For higher values of M, the success ratio plummeted

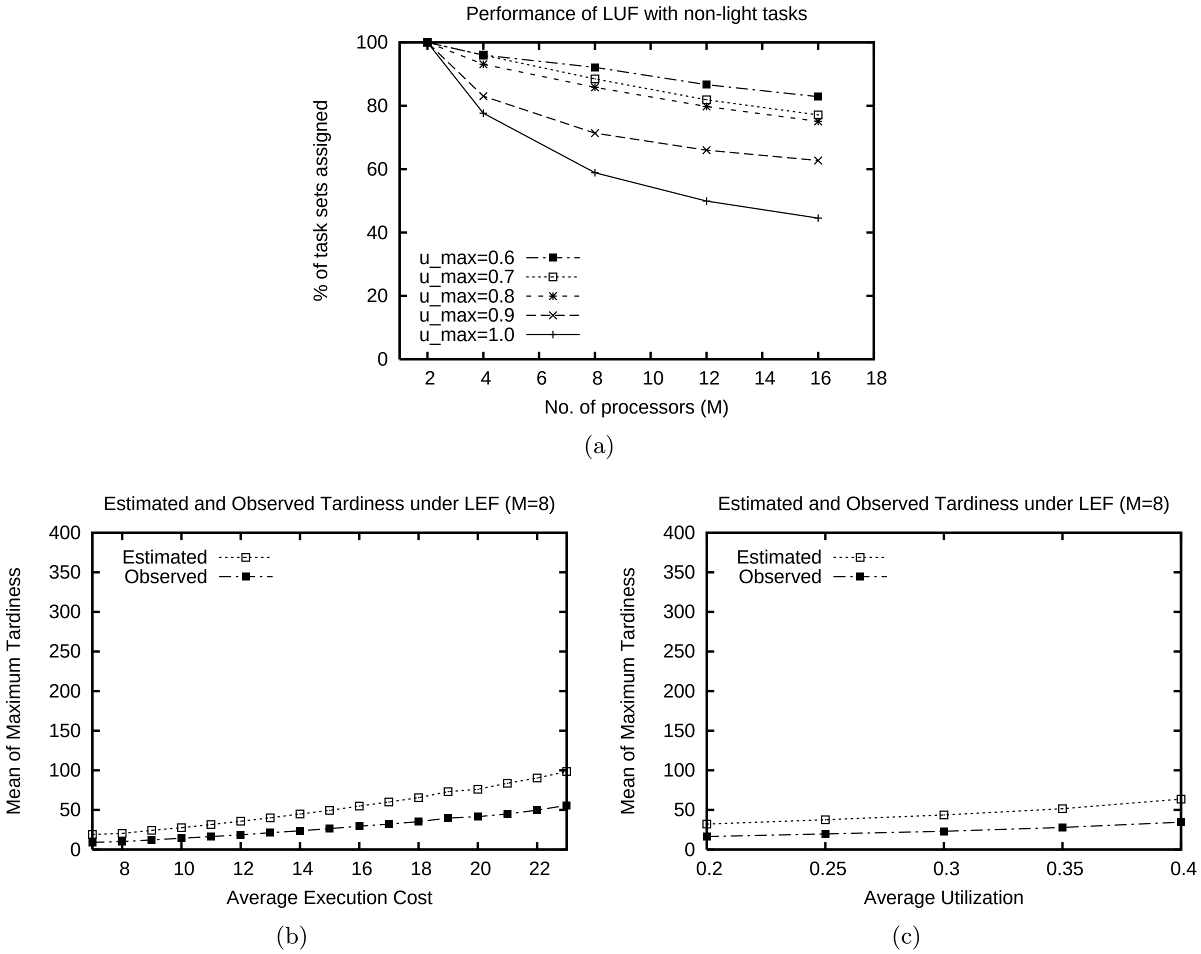
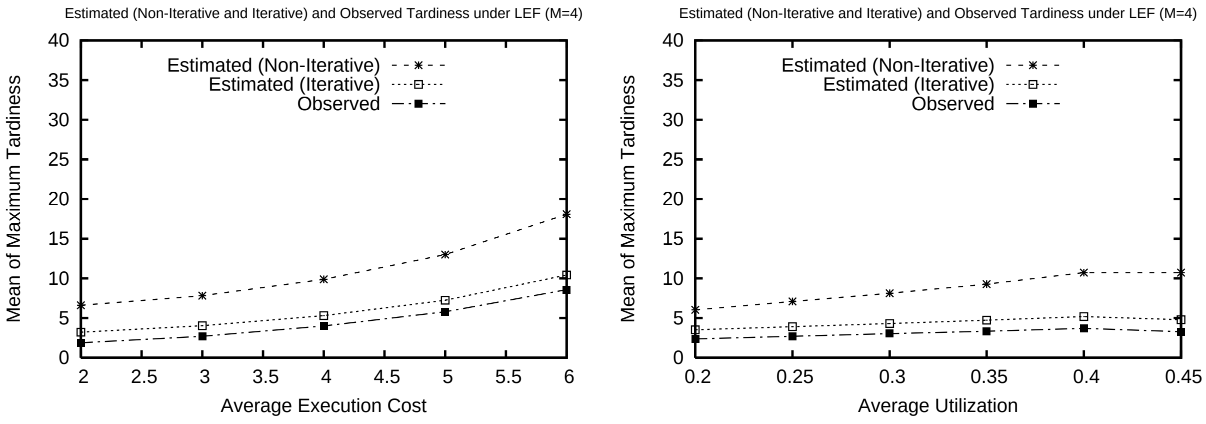


Figure 5.11: **(a)**Percentage of randomly-generated task sets with heavy tasks successfully assigned by the LUF heuristic. **(b) & (c)** Comparison of estimated and observed tardiness under EDF-fm-LEF by **(b)** average execution cost and **(c)** average utilization.

for all but the LUF approach. The percentage of task sets that LUF could successfully assign for varying M and $u_{\max}$ is shown in Figure 5.11(a). LEF performed next best (graphs not provided). However, even when $u_{\max} = 0.6$, its success percentage is approximately 79% when $M = 4$ and 24% when $M = 16$; the corresponding values are approximately 23.9% and 0.3%, respectively, when $u_{\max}$ is increased to 1.0. In this set of experiments also, HUF almost always performed worse than Random, and its success percentage was close to zero except when $M = 4$.

The third set of experiments was designed to evaluate the pessimism in the tardiness bound of (5.21). 300,000 task sets were generated with $u_{\max} = 0.5$ and $U_{sum} = 8$. The tardiness bound estimated by (5.21) under the LEF task assignment heuristic was computed for each



(a) (b)

Figure 5.12: Comparison of tardiness estimated by the iterative formulas to that estimated by the closed-form formula in (5.21) and observed tardiness under LEF task assignment by **(a)** average execution cost and **(b)** average utilization.

task set. A schedule under EDF-fm-LEF for 100,000 time units was also generated for each task set (when each task releases jobs in a synchronous, periodic manner) and the actual maximum tardiness observed was noted. (The time limit of 100,000 was determined by trial-and-error as an upper bound on the time within which tardiness converged for the tasks sets generated.) Plots of the average of the estimated and observed values for tasks grouped by e_{avg} and u_{avg} are shown in insets (b) and (c) of Figure 5.11, respectively. In general, we found that actual tardiness is only approximately half of the estimated value.

Finally, experiments were run to compare the bounds computed iteratively, which could require exponential time, to the actual tardiness. For this set of experiments, to facilitate computations, both periods and execution costs were chosen to be integers. Further, to reasonably constrain the length of the busy interval, the maximum period was restricted to 20, and some odd values such as 13, 17, and 19 were forbidden. Results are shown in Figure 5.12 and indicate that the bounds computed using this approach are very close to actual tardiness observed.

5.4 Summary

We have proposed an algorithm, EDF-fm, which is based upon EDF, for scheduling recurrent soft real-time task systems on multiprocessors, and have derived a tardiness bound that can be guaranteed under it. This algorithm places no restrictions on the total system utilization, but requires per-task utilizations to be at most one-half of a processor's capacity. This restriction is quite liberal, and hence, EDF-fm can be expected to be sufficient for scheduling a large percentage of soft real-time applications. Furthermore, under EDF-fm, at most $M - 1$ tasks may migrate, and each migrating task will execute on exactly two processors. Thus, task migrations are restricted and the migration overhead of EDF-fm is limited. We have also proposed heuristics for assigning tasks to processors and evaluated them, and proposed the use of the job-slicing technique, when possible, for significantly reducing the actual tardiness observed in practice. Finally, we have presented exponential-time formulas for computing more accurate tardiness bounds, which may be used during offline system design.

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

Chapter 6

A Schedulable Utilization Bound for EPDF[1]

In this chapter, we derive a schedulable utilization bound for the earliest-pseudo-deadline-first (EPDF) Pfair scheduling algorithm on multiprocessors. EPDF is not optimal but is less expensive than some other known Pfair algorithms. A motivation for determining a schedulable utilization bound for EPDF is provided in Section 6.1 below, which is followed by a derivation in Section 6.2. Proofs omitted from this chapter are provided in Appendix C.

6.1 Introduction and Motivation

In the previous two chapters, we considered scheduling soft real-time systems under preemptive and non-preemptive global EDF (g-EDF and g-NP-EDF), and EDF-fm, a restricted-migration algorithm based on partitioned EDF. Although g-EDF and g-NP-EDF can guarantee bounded tardiness to every task system that is feasible on M processors, their tardiness bounds can exceed the tolerance limits of some applications. EDF-fm also suffers from this drawback, apart from two others. First, EDF-fm cannot guarantee bounded tardiness to all feasible task systems with $u_{\max} > 0.5$. Second, its partitioning-based design, which makes it attractive to static task systems and task systems with large migration overheads, renders it unsuitable for dynamic task systems.

Pfair scheduling algorithms, which schedule tasks one quantum at a time, do not suffer from the above drawbacks. Though tasks may be prone to frequent migrations due to their

[1] Contents of this chapter previously appeared in preliminary form in the following paper:
[49] U. Devi and J. Anderson. Schedulable utilization bounds for EPDF fair multiprocessor scheduling. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications, Springer-Verlag Lecture Notes in Computer Science*, pages 261–280, August 2004.

quantum-based scheduling, as explained in Chapter 1, Pfair algorithms are still of interest in systems in which the gain in schedulability or the decrease in tardiness outweighs the loss in utilization due to migration overheads.

As discussed in Chapter 1, some restrictions and/or requirements of optimal Pfair algorithms can be overkill for soft real-time systems. If small tardiness bounds can be guaranteed under simpler algorithms in which one or more of those requirements are eliminated, then such algorithms may be sufficient for soft real-time systems. In this and the next chapter, we consider the *earliest-pseudo-deadline-first* (EPDF) algorithm, which is one such simpler, but non-optimal, Pfair algorithm.

EPDF is more efficient than optimal Pfair algorithms in that it does not use any tie-breaking rule to resolve ties among subtasks with the same pseudo-deadline, but disambiguates ties arbitrarily. PD^2, the most efficient of the known optimal Pfair algorithms, requires two tie-break parameters, the *b*-bit and group deadline, which were described in Chapter 3 in detail. Though these tie-break parameters can be computed for each subtask in constant time, there exist some soft and/or dynamic real-time systems in which not using them may still be preferable. Eliminating tie-breaking rules may also be preferable in embedded systems with slower processors or limited memory bandwidth.

The viability of EPDF for scheduling soft and/or dynamic real-time systems was first considered by Srinivasan and Anderson in [106], where they provide examples of such applications for which EPDF may be preferable to PD^2. Some web-hosting systems, server farms, packet processing in programmable multiprocessor-based routers, and packet transmission on multiple, parallel outgoing router links are among examples provided by them. In these systems, *fair* resource allocation is needed, so that quality-of-service guarantees can be provided. However, an extreme notion of fairness that precludes all deadline misses is not required. Moreover, in systems such as routers, the inclusion of tie-breaking information in subtask priorities may result in unacceptably high space overhead.

The applications mentioned above may also be dynamic in that the set of tasks and the utilizations of tasks requiring service may be in a constant flux. In [106], Srinivasan and Anderson also noted that the use of tie-breaking rules may be problematic for such dynamic task systems. As they explain, it is possible to *reweight* each task whenever its utilization changes such that its next subtask deadline is preserved. If no tie-breaking information is maintained, such an approach entails very little computational overhead. However, utilization changes can cause tie-breaking information to change, so if tie-breaking rules are used, reweighting may necessitate an $\mathcal{O}(N)$ cost for N tasks, due to the need to re-sort the scheduler's priority queue. This cost may be prohibitive if reallocations are frequent.

Motivated by the above reasons, Srinivasan and Anderson studied EPDF and they succeeded in showing that EPDF can guarantee a tardiness bound of one quantum for every subtask, provided a certain condition holds. Their condition can be ensured by limiting each task's weight to at most $1/2$, and can be generalized to apply to tardiness bounds higher than one. Unfortunately, Srinivasan and Anderson left open the question of whether such conditions are necessary to guarantee small constant tardiness.

We address the issues left open by Srinivasan and Anderson with respect to EPDF for soft real-time systems in the next chapter. Before doing so, in this chapter, we consider the following question, which is orthogonal to that addressed in prior work: If individual tasks cannot be subject to weight restrictions, then what would be a sufficient restriction on the total utilization of a task system for it to be scheduled correctly under EPDF? We answer this question by deriving a schedulable utilization bound for EPDF. The concept of schedulable utilization bound and its use were described in Chapter 1 in Section 1.4. Answering this question should help in identifying the conditions for which the soft real-time scheduling results of EPDF are applicable. Also, if the total utilization of an EPDF-scheduled soft real-time system is less than the schedulable utilization bound of EPDF, then that system can be treated as a hard real-time system.

In this chapter, we show that on M processors, EPDF can correctly schedule every task system τ with total utilization at most $\min(M, \frac{\lambda M(\lambda(1+W_{\max})-W_{\max})+1+W_{\max}}{\lambda^2(1+W_{\max})})$, where $W_{\max} = u_{\max}(\tau)$ and $\lambda = \max(2, \left\lceil \frac{1}{W_{\max}} \right\rceil)$. For $W_{\max} \geq \frac{1}{2}$, *i.e.*, $\lambda = 2$, this value reduces to $\frac{(2M+1)(2+W_{\max})-1}{4(1+W_{\max})}$, and as $W_{\max} \rightarrow 1.0$, it approaches $\frac{3M+1}{4}$, which, as $M \rightarrow \infty$, approaches $\frac{3M}{4}$, *i.e.*, 75% of the total processing capacity. When $W_{\max} > 1/2$, the utilization bound that we have derived for EPDF is greater than that of every known non-Pfair algorithm by around 25%. Currently, it is not known whether $\frac{3M+1}{4}$ is a worst-case, let alone, optimal utilization bound for EPDF. The closest that we have come in assessing the accuracy of our result is a counterexample that shows that when $W_{\max} \geq \frac{5}{6}$, worst-case schedulable utilization cannot exceed 86%. We have also considered extending this result for use in systems where tardiness may be permissible. This extension is presented along with the associated analysis in the next chapter.

6.2 A Schedulable Utilization Bound for EPDF

In this section, we derive a schedulable utilization bound for EPDF. The utilization bound that we derive is more accurate than that specified in Section 6.1, and can be used if the ρ values of tasks as defined in (6.1) can be computed.

Before getting started, we will define needed notation. Let ρ, $\rho_{\max}$, $W_{\max}$, and λ be defined as follows. (The task system τ will be omitted when it is unambiguous.)

$$\rho(T) \stackrel{\text{def}}{=} (T.e - \gcd(T.e, T.p))/T.p \tag{6.1}$$

$$\rho_{\max}(\tau) \stackrel{\text{def}}{=} \max_{T\in\tau}\{\rho(T)\} \tag{6.2}$$

$$W_{\max}(\tau) \stackrel{\text{def}}{=} \max_{T\in\tau}\{wt(T)\} \tag{6.3}$$

$$\lambda(\tau) \stackrel{\text{def}}{=} \max\left(2, \left\lceil \frac{1}{W_{\max}(\tau)} \right\rceil\right) \tag{6.4}$$

The utilization bound that we derive for EPDF is given by the following theorem.

Theorem 6.1 *Every GIS task system τ with total system utilization, $\sum_{T\in\tau} wt(T)$, at most $\min\left(M, \frac{\lambda M(\lambda(1+\rho_{\max})-\rho_{\max})+1+\rho_{\max}}{\lambda^2(1+\rho_{\max})}\right)$, where $\rho_{\max}$ and λ are as defined in (6.2) and (6.4), respectively, is correctly scheduled by* EPDF *on M processors.*

As a shorthand, we define $U(M, k, f)$ as follows.

Definition 6.1: $U(M, k, f) \stackrel{\text{def}}{=} \frac{kM(k(1+f)-f)+1+f}{k^2(1+f)}$.

We use a setup similar to that used by Srinivasan and Anderson in [105] in establishing the optimality of PD^2 to prove the above theorem. The overall strategy is to assume that the theorem is false, identify a task system that is minimal or smallest in the sense of having the smallest number of subtasks and that misses its deadline, and show that such a task system cannot exist. (It should be pointed out that although the setup is similar, the details of the derivation and proof are significantly different.)

If Theorem 6.1 does not hold, then there exist a $W_{\max} \leq 1$ and $\rho_{\max} < 1$, and a time t_d and a concrete task system τ defined as follows. (In these definitions, we assume that τ is scheduled on M processors.)

Definition 6.2: t_d is the earliest time at which any concrete task system with each task weight at most $W_{\max}$, $\rho(T)$ at most $\rho_{\max}$ for each task T, and total utilization at most

$\min(M, U(M, \lambda, \rho_{max}))$ misses a deadline under EPDF, *i.e.*, some such task system misses a subtask deadline at t_d, and no such system misses a subtask deadline prior to t_d.

Definition 6.3: τ is a task system with the following properties.

(T0) τ is concrete task system with each task weight at most $W_{\max}$, $\rho(T)$ at most $\rho_{\max}$, and total utilization at most $\min(M, U(M, \lambda, \rho_{\max}))$.

(T1) A subtask in τ misses its deadline at t_d in $\mathcal{S}$, an EPDF schedule for τ.

(T2) No concrete task system satisfying (T0) and (T1) releases fewer subtasks in $[0, t_d)$ than τ.

(T3) No concrete task system satisfying, (T0), (T1), and (T2) has a larger rank than τ at t_d, where the *rank* of a task system τ at t is the sum of the eligibility times of all subtasks with deadlines at most t, *i.e.*, $rank(\tau, t) = \sum_{\{T_i : T \in \tau \,\wedge\, \mathsf{d}(T_i) \le t\}} \mathsf{e}(T_i)$.

(T2) can be thought of as identifying a minimal task system in the sense of having a deadline miss with the fewest number of subtasks, subject to satisfying (T0) and (T1). It is easy to see that if Theorem 6.1 does not hold for all task systems satisfying (T0) and (T1), then some task system satisfying (T2) necessarily exists. (T3) further restricts the nature of τ by requiring subtask eligibility times to be spaced as much apart as possible.

The following shorthand notation will be used hereafter.

Definition 6.4: α denotes the total utilization of τ, expressed as a fraction of M, *i.e.*, $\alpha \stackrel{\text{def}}{=} \frac{\sum_{T \in \tau} wt(T)}{M}$.

Definition 6.5: $\delta \stackrel{\text{def}}{=} \frac{\rho_{\max}}{1+\rho_{\max}}$.

The lemma below follows from (T0) and the definition of α.

Lemma 6.1 $0 \le \alpha \le \frac{\min(M, U(M, \lambda, \rho_{\max}))}{M} \le 1$.

The next lemma is immediate from the definition of δ, (6.2), and (6.1). A proof is provided in Appendix C.

Lemma 6.2 $0 \le \delta < \frac{1}{2}$.

We now prove some properties about τ and $\mathcal{S}$. In proving some of these properties, we make use of the following three lemmas established in prior work by Srinivasan and Anderson.

(Before presenting these basic lemmas, it should be pointed out that some of these lemmas were established in the context of the PD^2 algorithm, but hold under EPDF as well. Furthermore, the way task system τ is defined, as in Definition 6.3, can vary with the problem at hand, and hence, the proofs of some such basic properties may have to be reworded to exactly hold for the task system under consideration. However, to avoid repetition, we simply borrow such properties without proof, but explicitly state that when we have done so.)

Lemma 6.3 (Srinivasan and Anderson [105]) *If* $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t)$*, then* $B(t) \neq \emptyset$*.*

The following is an intuitive explanation for why Lemma 6.3 holds. Recall from Section 3.6 that $B(t)$ is the set of all tasks that are active but not scheduled at t. Because $\mathsf{e}(T_i) \leq \mathsf{r}(T_i)$ holds, by Definition 3.1 and (3.15), only tasks that are active at t may receive positive allocations in slot t in the ideal schedule. Therefore, if every task that is active at t is scheduled at t, then the total allocation in $\mathcal{S}$ cannot be less than the total allocation in the ideal schedule, and hence, by (3.27), LAG cannot increase across slot t.

Lemma 6.4 (Srinivasan and Anderson [105]) *Let* $t < t_d$ *be a slot with holes and let* $T \in B(t)$*. Then, the critical subtask at* t *of* T *is scheduled before* t*.*

To see that the above lemma holds, let T_i be the critical subtask of T at t. By its definition, the IS-window of T_i overlaps slot t, but T is not scheduled at t. Also, there is at least a hole in t. Because EPDF does not idle a processor while there is a task with an outstanding execution request, T_i is thus scheduled before t.

Lemma 6.5 (Srinivasan and Anderson [105]) *Let* U_j *be a subtask that is scheduled in slot* t'*, where* $t' \leq t < t_d$ *and let there be a hole in* t*. Then* $\mathsf{d}(U_j) \leq t+1$*.*

This lemma is true because it can be shown that if $\mathsf{d}(U_j) > t+1$ holds, then U_j has no impact on the deadline miss at t_d. In other words, it can be shown that if the lemma does not hold, then the GIS task system obtained from τ by removing U_j also has a deadline miss at t_d, which is a contradiction to (T2).

Arguments similar to those used in proving the above lemma can be used to show the following. It is proved in Appendix C.

Lemma 6.6 *Let $t < t_d$ be a slot with holes and let U_j be a subtask that is scheduled at t. Then $\mathsf{d}(U_j) = t + 1$ and $\mathsf{b}(U_j) = 1$.*

Finally, we will also use the following lemma, which is a generalization of Lemma 6.3. It is also proved in Appendix C.

Lemma 6.7 *If $\mathsf{LAG}(\tau, t{+}1) > \mathsf{LAG}(\tau, t{-}\ell)$, where $0 \leq \ell \leq \lambda{-}2$ and $t \geq \ell$, then $B(t{-}\ell, t{+}1) \neq \emptyset$.*

The next lemma establishes some properties concerning $\mathcal{S}$.

Lemma 6.8 *The following properties hold for τ and $\mathcal{S}$.*

(**a**) *For all T_i in τ, $\mathsf{d}(T_i) \leq t_d$.*

(**b**) *Exactly one subtask of τ misses its deadline at t_d.*

(**c**) $\mathsf{LAG}(\tau, t_d) = 1$.

(**d**) *Let T_i be a subtask in τ that is scheduled at $t < t_d$ in $\mathcal{S}$. Then, $\mathsf{e}(T_i) = \min(\mathsf{r}(T_i), t)$.*

(**e**) $(\forall T_i \in \tau :: \mathsf{d}(T_i) < t_d \;\Rightarrow\; (\exists t :: \mathsf{e}(T_i) \leq t < \mathsf{d}(T_i) \;\wedge\; \mathcal{S}(T_i, t) = 1))$. *That is, every subtask with deadline before t_d is scheduled before its deadline.*

(**f**) *Let U_k be the subtask in τ that misses its deadline at t_d. Then, U is not scheduled in any slot in $[t_d - \lambda + 1, t_d)$.*

(**g**) *There is no hole in any of the last $\lambda - 1$ slots,* i.e., *in slots $[t_d - \lambda + 1, t_d)$.*

(**h**) *There exists a time $v \leq t_d - \lambda$ such that the following both hold.*

   (**i**) *There is no hole in any slot in $[v, t_d - \lambda)$.*

   (**ii**) $\mathsf{LAG}(\tau, v) \geq (t_d - v)(1 - \alpha)M + 1$.

(**i**) *There exists a time $u \in [0, v)$, where v is as defined in part (h), such that $\mathsf{LAG}(\tau, u) < 1$ and $\mathsf{LAG}(\tau, t) \geq 1$ for all t in $[u + 1, v]$.*

The proofs of parts (a)–(d) are similar to those formally proved in [105] for the optimal PD^2 Pfair algorithm. We give informal explanations here. Part (e) follows directly from Definition 6.2 and (T1). The rest are proved in Appendix C.

Proof of (a): This part holds because a subtask with deadline after t_d cannot impact the schedule for those with deadlines at most t_d. Therefore, even if all the subtasks with deadlines after t_d are removed, the deadline miss at t_d cannot be eliminated. This contradicts (T2).

Proof of (b): If several subtasks miss their deadlines at t_d, then even if all but one are removed, the remaining subtask will still miss its deadline, contradicting (T2).

Proof of (c): By part (a), all the subtasks in τ complete executing in the ideal schedule by t_d. Hence, the total allocation to τ in the ideal schedule up to t_d is exactly equal to the total number of subtasks in τ. By part (b), the total number of subtasks of τ scheduled in $\mathcal{S}$ in the same interval is fewer by exactly one subtask. Hence, the difference in allocations, $\mathsf{LAG}(\tau, t_d)$, is exactly one quantum.

Proof of (d): Suppose $\mathsf{e}(T_i)$ is not equal to $\min(\mathsf{r}(T_i), t)$. Then, by (3.12) and because T_i is scheduled at t, it is before $\min(\mathsf{r}(T_i), t)$. Hence, simply changing $\mathsf{e}(T_i)$ so that it equals $\min(\mathsf{r}(T_i), t)$ will not affect how T_i or any other subtask is scheduled. So, the deadline miss at t_d will persist. However, this change increases the rank of the task system, and hence, (T3) is contradicted. ■

Overview of the rest of the proof of Theorem 6.1. If t_d and τ as given by Definitions 6.2 and 6.3, respectively, exist, then by Lemma 6.8(i), there exists a time slot $u < v$, where v is as defined in Lemma 6.8(h), across which LAG increases to at least one. To prove Theorem 6.1, we show that for every such u, either **(i)** there exists a time u', where $u + 1 < u' \leq v$ or $u' = t_d$, such that $\mathsf{LAG}(\tau, u') < 1$, and thereby derive a contradiction to either Lemma 6.8(i) or Lemma 6.8(c), or **(ii)** v referred to above does not exist, contradicting Lemma 6.8(h). The crux of the argument we use in establishing the above is as follows. By Lemma 3.4, for LAG to increase across slot u, at least one hole is needed in that slot. We show that for every such slot, there are a sufficient number of slots without holes, and hence, that if (T0) holds, then the increase in LAG across slot u is offset by a commensurate decrease in LAG elsewhere that is sufficient to ensure that no deadline is missed.

In what follows, we state and prove several other lemmas that are required to accomplish this. We begin with a simple lemma that relates PF-window lengths to λ.

Lemma 6.9 *If $W_{\max} < 1$, then the length of the PF-window of each subtask of each task T in τ is at least λ; otherwise, it is at least $\lambda - 1$.*

Proof: Follows from the definition of λ in (6.4) and Lemma 3.1. ■

The next lemma involves subtask release times and deadlines and is proved in Appendix C.

Lemma 6.10 *For all $i \geq 1$, $k \geq 1$, the following holds.*

$$\mathsf{r}(T_{i+k}) \geq \begin{cases} \mathsf{d}(T_i) + k - 1, & \mathsf{b}(T_i) = 0 \\ \mathsf{d}(T_i) + k - 2, & \mathsf{b}(T_i) = 1 \end{cases}$$

The lemma that follows shows that LAG does not increase across the first $\lambda - 1$ slots, *i.e.*, across slots $0, 1, \ldots, \lambda - 2$.

Lemma 6.11 $(\forall t : 0 \leq t \leq \lambda - 2 :: \mathsf{LAG}(\tau, t+1) \leq \mathsf{LAG}(\tau, t))$.

Proof: Contrary to the statement of the lemma, assume that $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t)$ for some $0 \leq t \leq \lambda - 2$. Then, by Lemma 3.4, there is at least one hole in slot t, and by Lemma 6.3, $B(t)$ is not empty. Let U be a task in $B(t)$ and U_j its critical subtask at t. Because there is a hole in t, by Lemma 6.4,

(B) U_j is scheduled before t,

and hence, by Lemma 6.5, $\mathsf{d}(U_j) \leq t+1$ holds. We will consider two cases depending on $W_{\max}$. If $W_{\max} < 1$, then by Lemma 6.9, $|\omega(U_j)|$ (*i.e.*, the length of the PF-window of U_j) is at least λ, and hence, by Lemma 3.1, $\mathsf{r}(U_j) = \mathsf{d}(U_j) - |\omega(U_j)| \leq t + 1 - \lambda$ holds. By our assumption, $t < \lambda - 1$, and hence, $\mathsf{r}(U_j) < 0$, which is impossible. Thus, the lemma holds when $W_{\max} < 1$. On the other hand, if $W_{\max} = 1$, then, by (6.4), $\lambda = 2$ holds, and hence, by our assumption, $t = 0$. Therefore, by (B), U_j is scheduled before time zero, which is also impossible. The lemma follows. ∎

Lemma 3.4 showed that at least one hole is necessary in slot t for LAG to increase across t. The next lemma relates an increase in LAG to certain other conditions.

Lemma 6.12 *The following properties hold for* LAG *in* $\mathcal{S}$.

(a) *If* $\mathsf{LAG}(\tau, t+1, \mathcal{S}) > \mathsf{LAG}(\tau, t, \mathcal{S})$, *where* $\lambda - 1 \leq t < t_d - \lambda$, *then there is no hole in any slot in* $[t - \lambda + 2, t)$.

(b) *If* $\mathsf{LAG}(\tau, t+1, \mathcal{S}) > \mathsf{LAG}(\tau, t - \lambda + 2, \mathcal{S})$, *then there is no hole in slot* $t - \lambda + 1$.

Proof: We prove the two parts separately.

Proof of part (a). By the definition of λ in (6.4), $\lambda \geq 2$ holds. If $\lambda = 2$, then this part is vacuously true. Therefore, assume $\lambda \geq 3$, which, by (6.4), implies that

$$W_{\max} < \frac{1}{2}. \tag{6.5}$$

Because $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t)$, by Lemma 3.4,

(H) there is at least one hole in slot t,

and by Lemma 6.3, $B(t)$ is not empty. Let U be a task in $B(t)$ and let U_j be the critical subtask of U at t. By (H) and Lemma 6.4, U_j is scheduled before t. Let U_j be scheduled at $t' < t$, *i.e.*,

$$\mathcal{S}(U_j, t') = 1 \ \wedge \ t' < t. \tag{6.6}$$

Also, by Definition 3.2, $\mathsf{d}(U_j) \geq t+1$ holds, while by (H), (6.6), and Lemma 6.5, $\mathsf{d}(U_j) \leq t+1$ holds. Hence, we have

$$\mathsf{d}(U_j) = t + 1. \tag{6.7}$$

By (6.5) and Lemma 6.9, the length of U_j's PF-window, $|\omega(U_j)| \geq \lambda$, and hence, by (6.7) and the definition of PF-window in Lemma 3.1, $\mathsf{r}(U_j) \leq t - \lambda + 1$. We claim that U_j can be scheduled at ar after $t - \lambda + 2$. If there does not exist a predecessor for U_j, then U_j can be scheduled any time at or after $\mathsf{r}(U_j)$, and hence, at or after $t - \lambda + 1$. If not, let U_h be the predecessor of U_j. Then, $h \leq j$, and hence, by Lemma 6.10, $\mathsf{d}(U_h) \leq t - \lambda + 2$ holds. By Lemma 6.8(e), U_h does not miss its deadline, and hence, is scheduled at or before $t - \lambda + 1$. This implies that U_j is ready and can be scheduled at any time at or after $t - \lambda + 2$. Given this, if $t' > t - \lambda + 2$, then there is no hole in any slot in $[t - \lambda + 2, t')$. Hence, to complete the proof, it is sufficient to show that there is no hole in any slot in $[t', t)$. Assume to the contrary, and let v be the earliest slot with at least one hole in $[t', t)$. Then, by (6.6) and Lemma 6.5, $\mathsf{d}(U_j) \leq v + 1 \leq t$, which contradicts (6.7). Thus, v does not exist, and there is no slot with one or more holes in $[t', t)$.

Proof of part (b). By the statement of this part of the lemma, $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t - \lambda + 2)$. Hence, Lemma 6.7 applies with $\ell = \lambda - 2$. Therefore, $B(t - \lambda + 2, t + 1)$ is not empty. By definition, a task in $B(t - \lambda + 2, t + 1)$ is not scheduled anywhere in $[t - \lambda + 2, t + 1)$. Let U be a task in $B(t - \lambda + 2, t + 1)$ and let $t - \lambda + 2 \leq v \leq t$ be the latest slot in the interval $[t - \lambda + 2, t + 1)$ in which U is active, and let U_j be its critical subtask at v. Therefore, by

Definition 3.2,

$$\mathsf{d}(U_j) \geq v + 1 \geq t - \lambda + 3. \tag{6.8}$$

By Lemma 3.4, there is at least one hole in the interval $[t - \lambda + 2, t + 1)$. Hence, because U is in $B(t - \lambda + 2, t + 1)$, and thus is not scheduled anywhere in the interval concerned, U_j is scheduled at or before $t - \lambda + 1$ (as opposed to after the interval under consideration). Hence, if there is a hole in $t - \lambda + 1$, then by Lemma 6.5, it follows that $\mathsf{d}(U_j) \leq t - \lambda + 1$, which contradicts (6.8). Hence, there cannot be any hole in $t - \lambda + 1$, which completes the proof of this part. ∎

The next lemma bounds the lag of each task at time $t + 1$, where t is a slot with one or more holes.

Lemma 6.13 *If $t < t_d$ is a slot with one or more holes in $\mathcal{S}$. Then the following hold.*
(**a**) $(\forall T \in A(t) :: 0 \leq \mathsf{lag}(T, t + 1) \leq \rho(T))$
(**b**) $(\forall T \in B(t) :: \mathsf{lag}(T, t + 1) = 0)$
(**c**) $(\forall T \in I(t) :: \mathsf{lag}(T, t + 1) = 0)$

Proof: Part (c) has been proved formally in [106]. To see why part (c) holds, note that no task in $I(t)$ is scheduled at t. Because there is a hole in t, the latest subtask of a task in $I(t)$ with release time at or before t should have completed execution by t. Hence, such a task cannot be behind with respect to the ideal schedule.

Proof of part (a). Let T be a task in $A(t)$ and let T_i be its subtask scheduled at t. Then, in $\mathcal{S}$, T_i and all prior subtasks of T are scheduled in $[0, t+1)$, *i.e.*, each of these subtasks receives its entire allocation of one quantum in $[0, t + 1)$. Because there is a hole in t, by Lemma 6.6, we have

$$\mathsf{d}(T_i) = t + 1 \ \wedge \ \mathsf{b}(T_i) = 1. \tag{6.9}$$

By (3.17) and the final part of (3.15), this implies that T_i and all prior subtasks of T receive a total allocation of one quantum each in $[0, t + 1)$ in the ideal schedule also. Therefore, any difference in the allocations received in the ideal schedule and $\mathcal{S}$ is only due to subtasks that are released later than T_i. This is illustrated in Figure 6.1. Further, this difference can only be positive, and hence, the lower bound holds. We next show that the upper bound also holds.

Obviously, in $\mathcal{S}$, every subtask that is released later than T_i receives an allocation of zero in $[0, t+1)$. Therefore, the lag of T at $t+1$ is equal to the allocations that these later subtasks receive in the ideal schedule in the same interval. To determine this value, let T_j be the successor of T_i. By Lemma 6.10, if $j > i+1$, then $\mathsf{r}(T_j) \geq \mathsf{d}(T_i) = t+1$ holds. Hence, among the later subtasks, only T_{i+1} may receive a non-zero allocation in $[0, t+1)$. By (6.9), $\mathsf{b}(T_i) = 1$, and hence, by Lemma 6.10 again, $\mathsf{r}(T_{i+1})$ is at least $\mathsf{d}(T_i) - 1$, which, by (6.9), is at least t. Hence, in the ideal schedule, in the interval $[0, t+1)$, the allocation to T_{i+1} may be non-zero only in slot $\mathsf{r}(T_{i+1})$.

Thus, if $\mathsf{r}(T_{i+1}) \geq t+1$ or T_{i+1} is absent, then $\mathsf{lag}(T, t+1)$ is zero. Otherwise, it is given by the allocation that T_{i+1} receives in slot $t = \mathsf{r}(T_{i+1})$ in the ideal schedule. Because $\mathsf{b}(T_i) = 1$ holds, by Lemma 3.2, $\mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{r}(T_{i+1})) \leq \rho(T)$. Hence, $\mathsf{lag}(T, t+1) \leq \rho(T)$. This is illustrated in Figure 6.1.

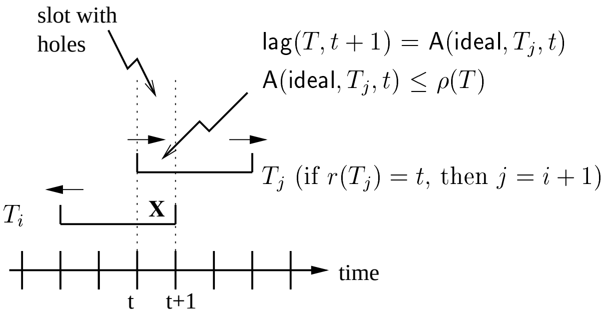


Figure 6.1: Lemma 6.13. PF-windows of a subtask T_i and its successor, T_j, are shown. If $r(T_j) = t$, then $j = i+1$ holds. T_i is scheduled in slot t (indicated by an "X"). There are one or more holes in t. Arrows over the window end-points indicate that the end-point could extend along the direction of the arrow. T_i and all prior subtasks of T complete executing at or before $t+1$. Therefore, the lag of T at $t+1$ is at most the allocation that T_{i+1} receives in slot t in the ideal schedule.

■

Proof of part (b). Let U be a task in $B(t)$ and let U_j be its critical subtask at t. Then, because there is a hole in t, by Lemma 6.4, U_j is scheduled before t. Thus, U_j and all prior subtasks of U complete executing by t in $\mathcal{S}$. Because U_j is U's critical subtask at t, U_j's successor is not eligible, and hence, is not released, before $t+1$. Further, by Lemma 6.5, $\mathsf{d}(U_j) \leq t+1$ holds. Hence, all subtasks of U released before $t+1$ complete executing by $t+1$ in both $\mathcal{S}$ and the ideal schedule. Therefore, lag for U at $t+1$ is zero. ■

The next lemma gives an upper bound on LAG at $t+1$ in terms of LAG at t and $t-\lambda+1$.

Lemma 6.14 *Let t, where $\lambda - 1 \leq t < t_d - \lambda$, be a slot with at least one hole. Then, $\mathsf{LAG}(\tau, t+1) \leq \mathsf{LAG}(\tau, t) \cdot \delta + \alpha M \cdot \delta$, and if there is no hole in any slot in $[t-\lambda+1, t)$, that is, there is no hole in any of the $\lambda - 1$ slots preceding t, then $\mathsf{LAG}(\tau, t+1) \leq \mathsf{LAG}(\tau, t-\lambda+1) \cdot \delta + (\lambda\alpha \cdot M - (\lambda-1)M) \cdot \delta$.*

Proof: By the statement of the lemma, there is at least one hole in slot t. Therefore, by Lemma 6.13, only tasks in $A(t)$, *i.e.*, tasks that are scheduled in slot t, may have a positive lag at $t+1$. Let x denote the number of tasks scheduled at t, *i.e.*, $x = \sum_{T \in \tau} \mathcal{S}(T, t) = |A(t)|$. Then, by (3.34), we have

$$
\begin{aligned}
\mathsf{LAG}(\tau, t+1) &\leq \textstyle\sum_{T\in A(t)} \mathsf{lag}(T, t+1) \\
&\leq \textstyle\sum_{T\in A(t)} \rho(T) && \text{(by Lemma 6.13)} \\
&\leq \textstyle\sum_{T\in A(t)} \rho_{\max} && \text{(by (6.2))} \\
&= |A(t)| \cdot \rho_{\max} \\
&= x \cdot \rho_{\max}. && (6.10)
\end{aligned}
$$

Using (3.29), $\mathsf{LAG}(\tau, t+1)$ can be expressed as follows.

$$
\begin{aligned}
\mathsf{LAG}(\tau, t+1) &\leq \mathsf{LAG}(\tau, t) + \textstyle\sum_{T\in\tau}(wt(T) - \mathcal{S}(T, t)) \\
&= \mathsf{LAG}(\tau, t) + \textstyle\sum_{T\in\tau} wt(T) - x && (\textstyle\sum_{T\in\tau}\mathcal{S}(T,t) = x) \\
&= \mathsf{LAG}(\tau, t) + \alpha M - x && \text{(by Def. 6.4)} \quad (6.11)
\end{aligned}
$$

By (6.10) and (6.11), we have

$$\mathsf{LAG}(\tau, t+1) \leq \min(x \cdot \rho_{\max}, \mathsf{LAG}(\tau, t) + \alpha M - x).$$

Because $\rho_{\max} \geq 0$, and hence, $x \cdot \rho_{\max}$ is non-decreasing with increasing x, whereas $\mathsf{LAG}(\tau, t) + \alpha M - x$ is decreasing, $\mathsf{LAG}(\tau, t+1)$ is maximized when $x \cdot \rho_{\max} = \mathsf{LAG}(\tau, t) + \alpha M - x$, *i.e.*, when $x = \mathsf{LAG}(\tau, t) \cdot \frac{1}{1+\rho_{\max}} + \alpha M \frac{1}{1+\rho_{\max}}$. Therefore, using either (6.10) or (6.11), we have

$$
\begin{aligned}
\mathsf{LAG}(\tau, t+1) &\leq \mathsf{LAG}(\tau, t) \cdot \left(\tfrac{\rho_{\max}}{1+\rho_{\max}}\right) + \alpha M \cdot \left(\tfrac{\rho_{\max}}{1+\rho_{\max}}\right) \\
&= \mathsf{LAG}(\tau, t) \cdot \delta + \alpha M \cdot \delta && \text{(by Def. 6.5).} \quad (6.12)
\end{aligned}
$$

By the statement of the lemma, $t \geq \lambda - 1$, and hence, $t - \lambda + 1 \geq 0$ holds. Therefore, using (3.31), $\mathsf{LAG}(\tau, t)$ can be expressed as follows.

$$\mathsf{LAG}(\tau, t) \leq \mathsf{LAG}(\tau, t-\lambda+1) + (\lambda - 1) \cdot \textstyle\sum_{T\in\tau} wt(T) - \sum_{u=t-\lambda+1}^{u=t-1} \sum_{T\in\tau} \mathcal{S}(T, u)$$

If there is no hole in any slot in $[t-\lambda+1, t)$, then $\sum_{T\in\tau} \mathcal{S}(T, u) = M$, for all u in $[t-\lambda+1, t)$. Hence,

$$
\begin{aligned}
\mathsf{LAG}(\tau, t) &\leq \mathsf{LAG}(\tau, t-\lambda+1) + (\lambda - 1) \cdot \textstyle\sum_{T\in\tau} wt(T) - \sum_{u=t-\lambda+1}^{u=t-1} \sum_{T\in\tau} \mathcal{S}(T, u) \\
&= \mathsf{LAG}(\tau, t-\lambda+1) + (\lambda - 1) \cdot \textstyle\sum_{T\in\tau} wt(T) - (\lambda - 1) \cdot M \\
&\leq \mathsf{LAG}(\tau, t-\lambda+1) + (\lambda - 1)\alpha \cdot M - (\lambda - 1)M && \text{(by Def. 6.4).} \quad (6.13)
\end{aligned}
$$

Substituting (6.13) in (6.12), we have $\mathsf{LAG}(\tau, t+1) \leq \mathsf{LAG}(\tau, t-\lambda+1) \cdot \delta + (\lambda\alpha \cdot M - (\lambda-1)M) \cdot \delta$.

■

The next two lemmas are purely mathematical and are proved in Appendix C. They will be used in proving a later lemma.

Lemma 6.15 *A solution to the recurrence*

$$\begin{array}{lcl} L_0 & < & \delta + \delta \cdot \alpha M \\ L_k & \leq & \delta \cdot L_{k-1} + \delta \cdot (\lambda \alpha M - (\lambda - 1)M), \end{array}$$

for all $k \geq 1$ and $0 \leq \delta < 1$, is given by

$$L_n < \delta^{n+1}(1 + \alpha M) + (1 - \delta^n)\left(\frac{\delta}{1-\delta}\right)(\lambda \alpha M - (\lambda - 1)M), \quad \textit{for all } n \geq 0.$$

Lemma 6.16 $\frac{M(\lambda-\delta-(\lambda-1)\delta^{n+1})+\delta^{n+2}-\delta^{n+1}+1-\delta}{M(\lambda-(\lambda-1)\delta^{n+1}-\delta^{n+2})} \geq \frac{M(\lambda-\delta)+\frac{1}{\lambda}}{\lambda M}$ *holds for all $n \geq 0, 0 \leq \delta \leq 1/2$, and $M \geq 1$.*

We will next show how to bound LAG at the end of an interval that does not contain λ consecutive slots without holes, after LAG increases to one at the beginning of the interval.

Lemma 6.17 *Let t, where $\lambda - 1 \leq t < t_d - \lambda$, be a slot across which* LAG *increases to at least one, i.e.,*

$$\lambda - 1 \leq t < t_d - \lambda \ \wedge \ \mathsf{LAG}(\tau, t) < 1 \ \wedge \ \mathsf{LAG}(\tau, t+1) \geq 1. \tag{6.14}$$

Let u, where $t < u \leq t_d - \lambda$, be such that there is at least one hole in at least one slot in every λ consecutive slots in the interval $[t+1, u)$. Then, $\mathsf{LAG}(\tau, u) < (\lambda - \lambda\alpha)M + 1$.

Proof: Because $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t)$ holds by (6.14), by Lemmas 3.4 and 6.12, we have (C1) and (C2) below, respectively. ((C2) holds by part (a) of Lemma 6.12 when $\lambda > 2$ and by part (b) of the same Lemma when $\lambda = 2$. By (6.4), $\lambda \geq 2$ holds.)

(C1) There is at least one hole in slot t.
(C2) There is no hole in slot $t - 1$.

By the statement of the lemma (that there is at least one hole in at least one slot in every λ consecutive slots in $[t+1, u)$) and (C1), there is at least one hole in at least one slot in every λ consecutive slots in $[t, u)$. Therefore, at most $\lambda - 1$ consecutive slots in the interval can be without any hole. Let

$$n \geq 0 \tag{6.15}$$

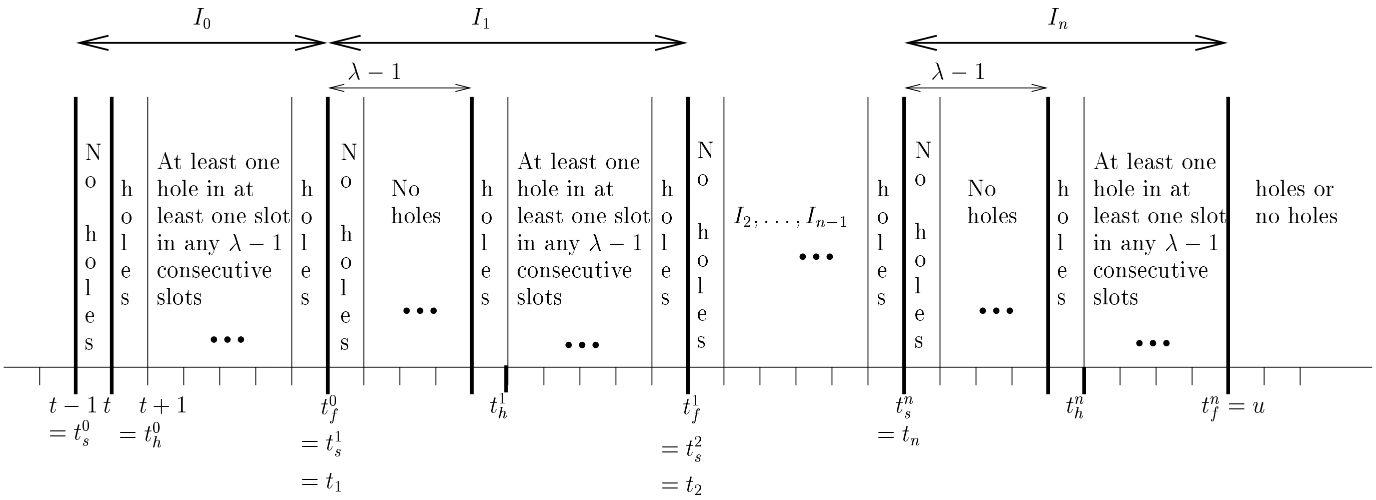


Figure 6.2: Lemma 6.17. $\mathsf{LAG}(\tau, t) < 1$ and $\mathsf{LAG}(\tau, t+1) \geq 1$. There is at least one hole in at least one slot in every λ consecutive slots in $[t-1, u)$. Each interval I_k, where $1 \leq k \leq n$, has one set of $\lambda - 1$ consecutive slots without holes. The objective is to determine an upper bound on $\mathsf{LAG}(\tau, u)$.

denote the number of slots in $[t, u)$ that each begin a block of $\lambda - 1$ consecutive slots with no holes. If $n > 0$, let $t_1, t_2, \ldots, t_n$, where $t_1 < t_2 < \ldots < t_n \leq u - \lambda + 1$ be such slots. By (C1), $t_1 > t$, and hence $t < t_1 < t_2 < \ldots < t_n \leq u - \lambda + 1$ holds. Further, $t_n + \lambda - 1 \leq u$ and there is no hole in $[t_i, t_i + \lambda - 1)$ for each i. Because there is at least one hole in at least one slot in any consecutive λ slots in $[t, u)$, there is at least one hole in $t_i - 1$ for all $1 \leq i \leq n$. Similarly, there is at least one hole in $t_i + \lambda - 1$ for all $1 \leq i \leq n-1$, and if $t_n + \lambda - 1 < u$ holds, then there is at least one hole in $t_n + \lambda - 1$. Also,

$$t_{i+1} - t_i \geq \lambda, \quad \text{for all } 1 \leq i \leq n-1. \tag{6.16}$$

We divide the interval $[t-1, u)$ into $n+1$ non-overlapping subintervals that each start at t or at one of the n slots that begin blocks of $\lambda - 1$ consecutive slots without holes, namely, $t_1, \ldots, t_n$, as shown in Figure 6.2. Note that $t - 1$ exists by the statement of the lemma, and by (C2), there is no hole in slot $t-1$. Also, as noted earlier, $t_1 > t$.

The subintervals denoted $I_0, I_1, \ldots, I_n$ are defined as follows.

$$I_0 \stackrel{\text{def}}{=} \begin{cases} [t-1, t_1), & \text{if } t_1 \text{ exists, } i.e., n > 0 \\ [t-1, u), & \text{otherwise} \end{cases} \tag{6.17}$$

$$I_n \stackrel{\text{def}}{=} [t_n, u), \qquad \text{if } n > 0 \tag{6.18}$$

$$I_k \stackrel{\text{def}}{=} [t_k, t_{k+1}), \quad \text{for all } 1 \le k < n \tag{6.19}$$

Before proceeding further, we introduce some more notation. We denote the start and end times of I_k, where $0 \le k \le n$, by t_s^k and t_f^k, respectively, *i.e.*, I_k is denoted as follows.

$$I_k \stackrel{\text{def}}{=} [t_s^k, t_f^k), \qquad \text{for all } k = 0, 1, \ldots, n \tag{6.20}$$

t_h^k is defined as follows for $0 \le k \le n$.

$$t_h^k \stackrel{\text{def}}{=} \begin{cases} t+1, & k = 0 \\ t_s^k + \lambda, & \text{otherwise} \end{cases} \tag{6.21}$$

LAG at t_h^k is denoted L_k, *i.e.*,

$$L_k \stackrel{\text{def}}{=} \mathsf{LAG}(\tau, t_h^k), \qquad \text{for all } k = 0, 1, \ldots, n \tag{6.22}$$

Note that each subinterval I_k, where $k \ge 1$, contains exactly one block of $\lambda - 1$ consecutive slots with no holes that begins at t_s^k and that the following holds (refer to Figure 6.2).

(C3) For all k, $0 \le k \le n$, there is **(i)** no hole in any slot in $[t_s^k, t_h^k - 1)$, **(ii)** at least one hole in $t_h^k - 1$, and **(iii)** at least one hole in at least one slot in $[\hat{t}, \hat{t} + \lambda - 1)$, where $t_h^k - 1 \le \hat{t} \le \min(t_f^k - 1, u - \lambda + 1)$.

Our goal now is to derive bounds for LAG at t_f^k, for all $0 \le k \le n$. Towards this end, we first establish the following claim, which states that LAG at any time in the interval $[t_h^k, t_f^k]$ is at most L_k for all k.

Claim 6.1 $(\forall k : 0 \le k \le n :: (\forall t' : t_h^k \le t' \le t_f^k :: \mathsf{LAG}(\tau, t') \le L_k))$.

For each k, the proof is by induction on t'.

Base Case: $t' = t_h^k$. The claim holds by the definition in (6.22).

Induction Step: Assuming that the claim holds at all times in the interval $[t_h^k, t']$, where $t_h^k \le t' < t_f^k$, we show that it holds at $t' + 1$. By this induction hypothesis, we have the following.

$$t_h^k \le t' < t_f^k \tag{6.23}$$

$$(\forall \hat{t} : t_h^k \le \hat{t} \le t' :: \mathsf{LAG}(\tau, \hat{t}) \le L_k) \tag{6.24}$$

We consider the following two cases.

Case 1: $t' \leq t_h^k + \lambda - 3$. For this case, $t' - \lambda + 2 \leq t_h^k - 1$ holds. By (C3)(ii), there is a hole in slot $t_h^k - 1$. By (6.23), $t_h^k - 1 < t'$. Thus, there is at least one hole in the interval $[t' - \lambda + 2, t')$, *i.e.*, in one of the $\lambda - 2$ slots immediately preceding t'. Therefore, by the contrapositive of Lemma 6.12(a), $\mathsf{LAG}(\tau, t'+1) \leq \mathsf{LAG}(\tau, t')$, which by (6.24), is at most L_k.

Case 2: $t' > t_h^k + \lambda - 3$. In this case,

$$t_h^k - 2 < t' - \lambda + 1 \tag{6.25}$$

holds. Because $t_f^k \leq t_f^n$ for all $0 \leq k \leq n$, we have $t_f^k \leq u$ by (6.17) and (6.18), when $n = 0$ and $n > 0$, respectively, and hence, by (6.23), we have $t' < u$. Therefore, $t' - \lambda + 1 < u - \lambda + 1$ holds. Also, because $\lambda \geq 2$, by (6.23), $t' - \lambda + 1 < t_f^k - 1$ follows. Thus, we have $t' - \lambda + 1 < \min(t_f^k - 1, u - \lambda + 1)$. Therefore, by (6.25), (C3)(iii) applies for $\hat{t} = t' - \lambda + 1$, and it follows that there is at least one hole in $[t' - \lambda + 1, t')$. If there is a hole in $[t' - \lambda + 2, t')$, then by the contrapositive of Lemma 6.12(a), $\mathsf{LAG}(\tau, t'+1) \leq \mathsf{LAG}(\tau, t')$, which by (6.24), is at most L_k. On the other hand, if there is no hole in $[t' - \lambda + 2, t')$, then there is a hole in $t' - \lambda + 1$ because there is at least one hole in $[t' - \lambda + 1, t')$. Thus, by the contrapositive of Lemma 6.12(b), $\mathsf{LAG}(\tau, t'+1) \leq \mathsf{LAG}(\tau, t' - \lambda + 2)$. By (6.25), $t' - \lambda + 2 \geq t_h^k$, and hence, by (6.24), $\mathsf{LAG}(\tau, t' - \lambda + 2) \leq L_k$. The claim follows for this case. ■

Having shown that $\mathsf{LAG}(\tau, t_f^k)$ is at most L_k, we now bound L_k. We start by determining a bound for L_0. By (6.22), $L_0 = \mathsf{LAG}(\tau, t_h^0)$, which by (6.21) is $\mathsf{LAG}(\tau, t+1)$. Thus,

$$\begin{aligned} L_0 &= \mathsf{LAG}(\tau, t+1) \\ &\leq \delta \cdot \mathsf{LAG}(\tau, t) + \delta \cdot \alpha M && \text{(by Lemma 6.14, because (C1) holds and by the statement of this lemma, } \lambda - 1 \leq t < t_d - \lambda) \\ &< \delta + \delta \cdot \alpha M, && (\mathsf{LAG}(\tau, t) < 1 \text{ by (6.14)}). \end{aligned} \tag{6.26}$$

We next determine an upper bound for L_k, where $1 \leq k \leq n$. Notice that by our definition of I_k in (6.19), we have $t_s^k = t_f^{k-1}$. Thus, $\mathsf{LAG}(\tau, t_s^k) = \mathsf{LAG}(\tau, t_f^{k-1})$, and hence, by Claim 6.1, we have

$$\mathsf{LAG}(\tau, t_s^k) \le L_{k-1}. \tag{6.27}$$

By (6.21), for $k > 0$, $t_h^k = t_s^k + \lambda$. By (6.16), $t_s^k + \lambda - 1 < t_s^{k+1}$. Also, $t_s^{k+1} \le u$ holds by (6.19), $k > 0$, and (6.18). By the statement of the lemma $u \le t_d - \lambda$. Thus,

$$t_s^k + \lambda - 1 < t_d - \lambda. \tag{6.28}$$

Also, $t_s^1 = t_1 > t$, and hence,

$$t_s^k + \lambda - 1 > t \ge \lambda - 1, \tag{6.29}$$

where the last inequality is from the statement of the lemma. By (C3)(ii), there is a hole in slot $t_h^k - 1 = t_s^k + \lambda - 1$, and by (C3)(i), no hole in any slot in $[t_s^k, t_h^k - 1)$, *i.e.*, in $[t_s^k, t_s^k + \lambda - 1)$. Therefore, because (6.28) and (6.29) also hold, Lemma 6.14 applies with $t_s^k + \lambda - 1$, and hence, $\mathsf{LAG}(\tau, t_s^k + \lambda) \le \delta \cdot \mathsf{LAG}(\tau, t_s^k) + \delta \cdot (\lambda\alpha \cdot M - (\lambda - 1) \cdot M)$, which by (6.22), (6.27), and $t_h^k = t_s^k + \lambda$ implies that

$$L_k = \mathsf{LAG}(\tau, t_h^k) = \mathsf{LAG}(\tau, t_s^k + \lambda) \le \delta \cdot L_{k-1} + \delta \cdot (\lambda\alpha \cdot M - (\lambda - 1) \cdot M), \qquad \text{for all } 1 \le k \le n. \tag{6.30}$$

By (6.18) and (6.20), we have $u = t_f^n$. Therefore, by Claim 6.1 and (6.15), we have

$$\mathsf{LAG}(\tau, u) = \mathsf{LAG}(\tau, t_f^n) \le L_n, \tag{6.31}$$

and hence, an upper bound on $\mathsf{LAG}(\tau, u)$ can be determined by solving the recurrence given by (6.26) and (6.30), which is restated below for convenience.

$$\begin{aligned} L_0 &< \delta + \delta \cdot \alpha M \\ L_k &\le \delta \cdot L_{k-1} + \delta \cdot (\lambda\alpha \cdot M - (\lambda - 1) \cdot M) \quad , k \ge 1 \end{aligned}$$

By Lemma 6.15, a solution to the above recurrence is given by

$$L_k < \delta^{k+1}(1 + \alpha M) + (1 - \delta^k)\left(\frac{\delta}{1 - \delta}\right)(\lambda\alpha \cdot M - (\lambda - 1) \cdot M).$$

Therefore, by (6.31), $\mathsf{LAG}(\tau, u) \le L_n < \delta^{n+1}(1 + \alpha M) + (1 - \delta^n)\left(\frac{\delta}{1-\delta}\right)(\lambda\alpha \cdot M - (\lambda - 1) \cdot M)$.

If L_n is at least $(\lambda - \lambda\alpha)M + 1$, then $\delta^{n+1}(1 + \alpha M) + (1 - \delta^n)\left(\frac{\delta}{1-\delta}\right)(\lambda\alpha \cdot M - (\lambda - 1) \cdot M) >$

$(\lambda - \lambda\alpha)M + 1$, which on rearranging terms implies that

$$\begin{aligned}
\alpha &> \frac{M(\lambda - \delta - (\lambda-1)\delta^{n+1}) + \delta^{n+2} - \delta^{n+1} + 1 - \delta}{M(\lambda - (\lambda-1)\delta^{n+1} - \delta^{n+2})} \\
&\geq \frac{M(\lambda - \delta) + \frac{1}{\lambda}}{\lambda M} && \text{(by Lemma 6.16, because } 0 \leq \delta < 1/2 \text{ by Lemma 6.2)} \\
&= \frac{\lambda M(\lambda(1+\rho_{\max}) - \rho_{\max}) + 1 + \rho_{\max}}{\lambda^2 M(1+\rho_{\max})} && \text{(by Def. 6.5)} \\
&= \frac{U(M, \lambda, \rho_{\max})}{M} && \text{(by Def. 6.1)} \\
&\geq \frac{\min(M, U(M, \lambda, \rho_{\max}))}{M}. && (6.32)
\end{aligned}$$

Because (6.32) is in contradiction to Lemma 6.1, we conclude that $L_n < (\lambda - \lambda\alpha)M + 1$. Hence, by (6.31), $\mathsf{LAG}(\tau, u) \leq L_n < (\lambda - \lambda\alpha)M + 1$. ■

Lemma 6.18 *Let $\lambda - 1 \leq t < t_d - \lambda$ be a slot such that $\mathsf{LAG}(\tau, t) < 1 \ \wedge \ \mathsf{LAG}(\tau, t+1) \geq 1$ and let u be the earliest time after t such that $u = t_d - \lambda$ or there is no hole in any slot in $[u, u+\lambda)$. (Note that this implies that there is at least one hole in at least one slot in every λ consecutive slots in $[t+1, u)$.) Then, at least one of the following holds.*

$t < u \leq t_d - 2\lambda$, there is a hole in $u-1$, $(\forall \hat{t} : u \leq \hat{t} < u + \lambda :: \mathsf{LAG}(\tau, \hat{t}) < (\lambda - \lambda\alpha)M + 1)$, and $\mathsf{LAG}(\tau, u+\lambda) < 1$. (6.33)

$\mathsf{LAG}(\tau, t_d) < 1$. (6.34)

There exists at least one slot with a hole before $t_d - \lambda$ and $(\forall v : t' + 1 \leq v \leq t_d - \lambda :: \mathsf{LAG}(\tau, v) < (t_d - v) \cdot (1-\alpha)M + 1)$, where t' is the latest slot with a hole before $t_d - \lambda$. (6.35)

Proof: Because $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t)$ holds (by the statement of the lemma), by Lemma 3.4, we have the following.

(D1) There is at least one hole in t.

We consider two cases depending on u. (By the definition of u in the statement of the lemma, if $u < t_d - \lambda$ holds, then there is no hole in u.)

Case 1: $u \leq t_d - \lambda$ and there is no hole in u. By Lemma 6.8(g),

(D2) there is no hole in any slot in $[t_d - \lambda + 1, t_d)$.

If $u = t_d - \lambda$, then by the condition of this case, there is no hole in $t_d - \lambda$, and hence, by (D2), in any slot in $[t_d - \lambda, t_d)$, *i.e.*, in $[u, u + \lambda)$. On the other hand, if $u < t_d - \lambda$, then by the definition of u, there is no hole in $[u, u + \lambda)$. Thus, in both cases, we have the following.

(D3) There is no hole in any slot in $[u, u + \lambda)$.

By the definition of u, there is at least one hole in at least one slot in any λ consecutive slots in the interval $[t + 1, u)$. Therefore, by Lemma 6.17, we have

$$\mathsf{LAG}(\tau, u) < (\lambda - \lambda\alpha)M + 1. \tag{6.36}$$

To prove the lemma for this case, we consider the following two subcases.

Subcase 1(a): $\boldsymbol{u \leq t_d - 2\lambda}$. For this subcase, we first show that

(D4) there is at least one hole in $u - 1$.

Because (D3) and $u < t_d - \lambda$ (which is implied by the condition of this subcase, since $\lambda \geq 2$) hold, the absence of holes in $u-1$ would imply that there is no hole in any slot in $[u-1, u+\lambda)$. This is a contradiction of the fact that u is the earliest time after t such that either there is no hole in any slot in $[u, u + \lambda)$ or $u = t_d - \lambda$. Thus, there is at least one hole in $u - 1$.

We next show that $\mathsf{LAG}(\tau, u + \lambda) < 1$ holds. By (3.31), we have

$$\begin{aligned}
&\mathsf{LAG}(\tau, u + \lambda) \\
&\quad \leq \quad \mathsf{LAG}(\tau, u) + \lambda \cdot \textstyle\sum_{T \in \tau} wt(T) - \sum_{v=u}^{v=u+\lambda-1} \sum_{T \in \tau} \mathcal{S}(T, v) \\
&\quad = \quad \mathsf{LAG}(\tau, u) + \lambda \cdot \textstyle\sum_{T \in \tau} wt(T) - \lambda M \qquad \text{(by (D3), there is no hole in } [u, u + \lambda)) \\
&\quad < \quad (\lambda - \lambda\alpha)M + 1 + \lambda \cdot \textstyle\sum_{T \in \tau} wt(T) - \lambda M \qquad \text{(by (6.36), } \mathsf{LAG}(\tau, u) < (\lambda - \lambda\alpha)M + 1) \\
&\quad = \quad (\lambda - \lambda\alpha)M + 1 + \lambda\alpha M - \lambda M \qquad \text{(by Def. 6.4)} \\
&\quad = \quad 1.
\end{aligned} \tag{6.37}$$

Further, by (D3), there is no hole in any slot in $[u, u + \lambda)$, hence, by Lemma 3.4, LAG cannot increase across any of these slots. Therefore, by (6.36), $(\forall \hat{t} : u < \hat{t} < u + \lambda :: \mathsf{LAG}(\tau, \hat{t}) \leq \mathsf{LAG}(\tau, u) < (\lambda - \lambda\alpha)M + 1)$ holds, which together with (D4), (6.36) and (6.37), establishes condition (6.33) for this subcase.

Subcase 1(b): $\boldsymbol{t_d - 2\lambda < u \leq t_d - \lambda}$. For this subcase, we first show that there is no hole in any slot in $[u, t_d)$. If $u = t_d - \lambda$, then, by (D3), there is no hole in $[u, t_d)$. Otherwise, because $u > t_d - 2\lambda$ holds, we have $u + \lambda \geq t_d - \lambda + 1$. Thus, by (D3), there is no hole in any slot in $[u, t_d - \lambda + 1)$, and hence, by (D2), no hole in $[u, t_d)$. By (3.30), $\mathsf{LAG}(\tau, t_d) \leq \mathsf{LAG}(\tau, u) + (t_d - u) \cdot \sum_{T \in \tau} wt(T) - \sum_{v=u}^{t_d-1} \mathsf{A}(\mathcal{S}, \tau, v)$. Therefore, by Definition 6.4, $\mathsf{LAG}(\tau, t_d) \leq \mathsf{LAG}(\tau, u) + (t_d - u)\alpha M - \sum_{v=u}^{t_d-1} \mathsf{A}(\mathcal{S}, \tau, v)$. Since there is no hole in $[u, t_d)$ in $\mathcal{S}$, $\mathsf{A}(\mathcal{S}, \tau, v) = M$, for every v in $[u, t_d)$. Therefore, $\mathsf{LAG}(\tau, t_d) \leq \mathsf{LAG}(\tau, u) + (t_d - u)(\alpha M - M)$, which by (6.36), implies $\mathsf{LAG}(\tau, t_d) < (\lambda - \lambda\alpha)M + 1 + (t_d - u)(\alpha M - M)$. Since, by the condition of this case, $u \leq t_d - \lambda$ holds, we have $t_d - u \geq \lambda$. Hence, because $\alpha \leq 1$ (by Lemma 6.1), $\mathsf{LAG}(\tau, t_d) < 1$ holds. Thus, (6.34) holds for this subcase.

Case 2: $\boldsymbol{u = t_d - \lambda}$ and there is a hole in slot $\boldsymbol{t_d - \lambda}$. Let u' denote the latest slot with at least one hole in $[t, t_d - \lambda)$. Because (D1) holds, u' exists, and the following holds.

(D5) $u' < t_d - \lambda$ and there is no hole in any slot in $[u' + 1, t_d - \lambda)$.

Because no λ consecutive slots in $[t, t_d - \lambda)$ are without holes (implied by the statement of the lemma, since $u = t_d - \lambda$ for this case), applying Lemma 6.17 (with $u = u' + 1$), we have

$$\mathsf{LAG}(\tau, u' + 1) < \lambda(1 - \alpha)M + 1. \tag{6.38}$$

By (D5), $u' < t_d - \lambda$, and hence, $u' + 1 \leq t_d - \lambda$ holds. Therefore, $\lambda \leq (t_d - (u' + 1))$, and hence, because $\alpha \leq 1$ by Lemma 6.1,

$$\lambda(1 - \alpha)M \leq (t_d - (u' + 1))(1 - \alpha)M \tag{6.39}$$

holds. Therefore, by (6.38), we have

$$\begin{aligned} \mathsf{LAG}(\tau, u' + 1) &< \lambda(1 - \alpha)M + 1 && (6.40)\\ &\leq (t_d - (u' + 1))(1 - \alpha)M + 1 \qquad \text{(by (6.39))}. && (6.41) \end{aligned}$$

Hence, for all $\hat{t}$, where $u' + 2 \leq \hat{t} \leq t_d - \lambda$, by (3.31), we have the following.

$$\begin{aligned} &\mathsf{LAG}(\tau, \hat{t})\\ &\leq \mathsf{LAG}(\tau, u' + 1) + (\hat{t} - u' - 1) \cdot \textstyle\sum_{T \in \tau} wt(T) - \sum_{v=u'+1}^{\hat{t}-1} \sum_{T \in \tau} \mathcal{S}(T, v)\\ &< (\lambda - \lambda\alpha)M + 1 + (\hat{t} - u' - 1) \cdot \textstyle\sum_{T \in \tau} wt(T) - \sum_{v=u'+1}^{\hat{t}-1} \sum_{T \in \tau} \mathcal{S}(T, v) \text{ (by (6.40))}\\ &= (\lambda - \lambda\alpha)M + 1 + (\hat{t} - u' - 1) \cdot \textstyle\sum_{T \in \tau} wt(T) - (\hat{t} - u' - 1) \cdot M \end{aligned}$$

$$
\begin{aligned}
& && \text{(by (D5) and the definition of } \hat{t}, \\
& && \text{there in no hole in } [u'+1,\hat{t})) \\
&= (\lambda - \lambda\alpha)M + 1 + \alpha(\hat{t} - u' - 1)\cdot M - (\hat{t} - u' - 1)\cdot M && \text{(by Def. 6.4)} \\
&\leq (\lambda - \lambda\alpha)M + 1 && \text{(by Lemma 6.1)} \\
&\leq (t_d - \hat{t})\cdot(1-\alpha)M + 1 && \text{(because } \hat{t} \leq t_d - \lambda) \qquad (6.42)
\end{aligned}
$$

By the definition of u', (6.41), the definition of $\hat{t}$, and (6.42), condition (6.35) holds for this case. ■

By part (i) of Lemma 6.8, there exists a t, where $0 \leq t < v$, where v is as defined in Lemma 6.8(h), such that $\mathsf{LAG}(\tau, t) < 1$ and $\mathsf{LAG}(\tau, s) \geq 1$, for all s in $[t+1, v]$. Since by Lemma 6.11, $t \geq \lambda - 1$ holds, Lemma 6.18 applies for t. Let $t' = u + \lambda$, where u is as defined in Lemma 6.18. If (6.33) holds, then

(E) $u \leq t_d - 2\lambda$ and there is a hole in $u - 1$,

and $\mathsf{LAG}(\tau, t') < 1$ holds. Hence, if $t' < v$ holds, then the existence of t, and hence Lemma 6.8(i), is contradicted. On the other hand, if $t' \geq v$ holds, then, because, by (E), there is a hole in $u - 1 < t_d - 2\lambda$, and by the definition of v in Lemma 6.8(h), which states that there is no hole in any slot in $[v, t_d - \lambda)$, we have $u - 1 < v$, *i.e.*, $v \geq u$. Therefore, we have, $u \leq v \leq t'$. However, by (6.33), LAG at every time between u and $u + \lambda - 1 = t' - 1$ is less than $(\lambda - \lambda\alpha)M + 1$, and at time $u + \lambda = t'$ is less than one, and hence, is less than $(\lambda - \lambda\alpha)M + 1$ (as $\alpha \leq 1$). This contradicts Lemma 6.8(h). If (6.34) holds, then Lemma 6.8(c) is contradicted. Finally, if (6.35) holds, then at every time s after the latest slot with a hole before $t_d - \lambda$ and until $t_d - \lambda$, $\mathsf{LAG}(\tau, s) < (t_d - s)\cdot(1-\alpha)M + 1$, which contradicts part (h) of Lemma 6.8. Therefore, our assumption that τ misses a deadline at t_d is incorrect, which in turn proves Theorem 6.1.

According to Theorem 6.1, $\min(M, U(M, \lambda, \rho_{\max}))$ is a schedulable utilization bound for EPDF on M processors. We now show that every GIS task system with total utilization at most $\min(M, U(M, \lambda, W_{\max}))$ is also correctly scheduled under EPDF. For this, first note that the first derivative of $U(M, \lambda, f) = \frac{\lambda M(\lambda(1+f)-f)+1+f}{\lambda^2(1+f)}$ with respect to f is given by $\frac{\lambda^4 M^2(f-1)-\lambda^3 M^2}{\lambda^2 M(1+f)}$, which is negative for all $M \geq 1$, $\lambda \geq 1$, and $f \leq 1$. Hence, $U(M, \lambda, f)$ is a decreasing function of f for $f \leq 1$. By (6.1), $\rho(T) < wt(T)$ holds for each task T. Therefore, $0 \leq \rho_{\max} < W_{\max} \leq 1$ holds, and hence, $U(M, \lambda, W_{\max}) < U(M, \lambda, \rho_{\max})$ holds, by which we have the following corollary to Theorem 6.1.

Corollary 6.1 *A GIS task system τ is schedulable on M processors under* EPDF *if the total*

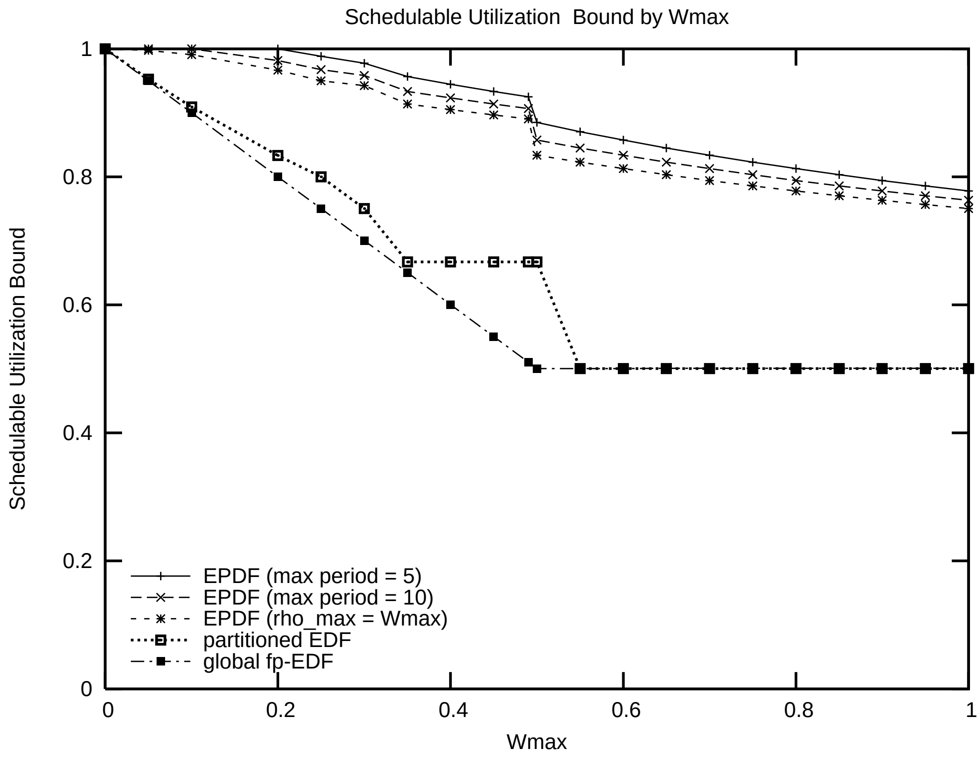


Figure 6.3: Schedulable utilization bound (expressed as a fraction of the total available processing capacity) by $W_{\max}$.

utilization of τ is at most $\min(M, \frac{\lambda M(\lambda(1+W_{\max})-W_{\max})+1+W_{\max}}{\lambda^2(1+W_{\max})})$, where $W_{\max}$ and λ are as defined in (7.10) and (6.4), respectively.

Comparison with partitioned EDF and a variant of global EDF. Figure 6.3 shows how the utilization bound derived for EPDF compares to the best known bounds of partitioned EDF [87] and a variant of global EDF, called fp-EDF. Under fp-ED, tasks with utilizations greater than 1/2 are statically given higher priority than the remaining tasks, and the remaining tasks are scheduled using EDF [22]. In general, there does not exist a correlation between $W_{\max}$ and $\rho_{\max}$. That is, $W_{\max}(\tau) > W_{\max}(\tau') \not\Rightarrow \rho_{\max}(\tau) > \rho_{\max}(\tau')$, where τ and τ' are two distinct task systems. Thus, a general characterization of any utilization bound as a function of $\rho_{\max}$ may not convey useful information. Hence, in Figure 6.3, each bound is plotted as a function of $W_{\max}$ only. Also, $\rho_{\max}(\tau)$ cannot be determined if only $W_{\max}(\tau)$ is known. However, since $\gcd(T.e, T.p) \geq 1$ holds for every T, by (6.1) and (6.2), $\rho_{\max}(\tau)$ is at most $W_{\max}(\tau) - 1/p_{\max}(\tau)$, where $p_{\max}$ is the maximum period of any task. Hence, for an arbitrary task system, a utilization bound higher than that given by $\min(M, U(M, \lambda, W_{\max}))$ is possible if at least $p_{\max}$ is also known. Thus, in Figure 6.3, utilization bounds for EPDF for tasks systems with $p_{\max} = 5$ and $p_{\max} = 10$ computed using Theorem 6.1 assuming $\rho_{\max} = W_{\max} - 1/p_{\max}$ are plotted, as is the $U(M, \lambda, W_{\max})$ bound. As can be seen from the plots, the utilization bound of EPDF is higher than those of the other two algorithms by around 25% in the worst case. Also, the utilization bound of EPDF is piece-wise linear with discontinuities whenever λ changes, *i.e.*, at $W_{\max} = 1/p$, where p is an integer.

In [107], Srinivasan and Anderson have shown that EPDF can correctly schedule every task system with $W_{\max} \leq \frac{1}{M-1}$. With $W_{\max} = \frac{1}{M-1}$, the utilization bound given by Corollary 6.1 is $M - \frac{1}{M-1} + \frac{1}{(M-1)^2}$, which approaches M asymptotically. The discrepancy for lower values of M is due to the approximation of $\rho_{\max}$ by $W_{\max}$. Hence, the results of this paper are useful only when $W_{\max} > \frac{1}{M-1}$ holds.

How accurate is the utilization bound given by Theorem 6.1? Is it the worst-case for EPDF? As yet, we do not fully know the answers to these questions. The closest we have come towards providing any answer is the following task system, which can miss deadlines under EPDF.

Counterexample. Consider a task system that consists of $2n+1$ tasks of weight $\frac{1}{2}$, n tasks of weight $\frac{3}{4}$, and n tasks of weight $\frac{5}{6}$ scheduled on $3n$ processors, where $n \geq 2$. There is an EPDF schedule for this task system in which a deadline miss occurs at time 12. One such schedule is shown in Figure 6.4. The total utilization of this task system is $\frac{31n}{12} + \frac{1}{2}$, which approaches 86.1% of $3n$, the total processing capacity, as n approaches infinity. The utilization bound given by Theorem 6.1 for a task system with task parameters as in this example is slightly above 80% for large M. Hence, at least asymptotically, the value derived does not seem to be too pessimistic. However, the same cannot be said when M is small. For instance, for the example under consideration, the total utilizations when $n = 2$ and $n = 3$ (*i.e.*, $M = 6$ and $M = 9$) are 94.4% and 91.6%, respectively, while the estimated values are only 84.1% and 82.7%. Further, though this single task system is too little data to draw general conclusions, the difficulty in identifying counterexamples and some of the approximations made in the analysis make us strongly believe that our result is not tight.

6.3 Summary

We have determined a utilization bound for the non-optimal earliest-pseudo-deadline-first (EPDF) Pfair scheduling algorithm on multiprocessors, and thereby, presented a sufficient

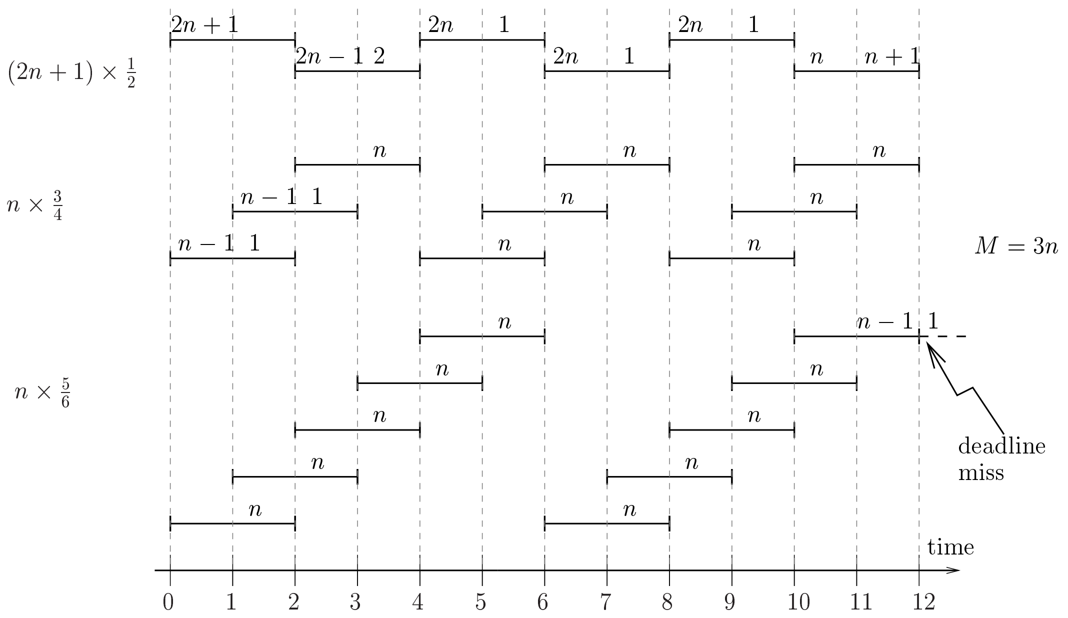


Figure 6.4: An EPDF schedule with a deadline miss for a task system with $2n+1$ tasks of weight $\frac{1}{2}$, n tasks with weight $\frac{3}{4}$, and n tasks with weight $\frac{5}{6}$ on $M = 3n$ processors, where $n \geq 2$. The number of tasks for each weight scheduled in each time slot is indicated. As $n \to \infty$, the total utilization of this task system approaches 86.1%.

schedulability test for EPDF. In general, on $M > 2$ processors, the utilization bound derived is at least $\frac{3M+1}{4}$, and is higher than that of every non-Pfair algorithm by around 25% when task utilizations are not restricted. Our bound is expressed as a decreasing function of the maximum weight, $W_{\max}$, of any task, and hence, if this value is known, may be used to schedule task systems with total utilization exceeding $\frac{3M+1}{4}$. A knowledge of EPDF's utilization bound should help in identifying the task systems for which the soft real-time scheduling results of EPDF are applicable. Also, if the total utilization of an EPDF-scheduled soft real-time system is less than the schedulable utilization bound of EPDF, then that system can be treated as a hard real-time system.

Currently, it is not known whether the utilization bound derived is a worst-case schedulable utilization for EPDF. We have only presented a counterexample that shows that the worst-case value for $W_{\max} \leq \frac{5}{6}$ is at most 86.1%, which also suggests that a significant improvement to the result may not be likely. Towards the end of the next chapter, we extend this result to allow non-zero tardiness.

Chapter 7

Improved Conditions for Bounded Tardiness under EPDF[1]

In this chapter, we present results concerning EPDF that can enable its use with more soft real-time systems than previously possible. Our contributions are as follows.

As mentioned in Chapter 6, the efficacy of EPDF for soft real-time systems was first considered by Srinivasan and Anderson in [106]. They conjectured that on M processors, EPDF can ensure a tardiness bound of one quantum to every task system with total utilization at most M (*i.e.*, every task system that is feasible on M processors). Our first result in this chapter refutes this conjecture. For this, we present counterexamples that show that tardiness under EPDF can not only exceed one quantum, but can be up to four quanta. The method used to generate these counterexamples suggests that, in general, EPDF cannot guarantee small constant tardiness that is independent of the task system parameters.

In [106], Srinivasan and Anderson showed that EPDF can guarantee a tardiness bound of $q \geq 1$ quanta for every ‘ subtask of a feasible task system, provided a certain condition holds. Their condition can be ensured by limiting each task's weight to at most $\frac{q}{q+1}$. As a second contribution, we improve upon this result, and show that a more liberal restriction of $\frac{q+2}{q+3}$ on the weight of each task is sufficient to ensure that tardiness does not exceed q quanta. Note that, when $q = 1$, our result presents an improvement of 50% over the previous result.

As a third contribution, we present a sufficient restriction of $U \stackrel{\text{def}}{=}$

[1]Contents of this chapter previously appeared in preliminary form in the following paper:
[48] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. In *Proceedings of the 12th International Workshop on Parallel and Distributed Real-Time Systems*, April 2004. 8 pages (on CD-ROM).

$\min(M, \frac{((q+1)W_{\max}+(q+2))M+((2q+1)W_{\max}+1)}{2(q+1)W_{\max}+2})$ on the total utilization of a task system for ensuring a tardiness bound of $q \geq 1$ quanta on M processors, for task systems whose maximum task weight may exceed $\frac{q+2}{q+3}$. For $q = 1$, U is around 83.5%, which is higher by around 8% over that presented in the previous chapter for hard real-time systems.

The results described above are presented in order in Sections 7.1, 7.2, and 7.3. Section 7.4 concludes.

7.1 Counterexamples

It is easy to show that subtask deadlines can be missed under EPDF. In [106], it was conjectured that EPDF ensures a tardiness bound of one for every feasible task system. We now show that this conjecture is false.

Theorem 7.1 *Tardiness under* EPDF *can exceed three quanta for feasible GIS task systems. In particular, if* EPDF *is used to schedule task system* $\tau^{(i)}$ $(1 \leq i \leq 3)$ *in Table 7.1, then a tardiness of* $i + 1$ *quanta is possible.*

Proof: Figure 7.1 shows a schedule for $\tau^{(1)}$, in which a subtask has a tardiness of two at time 50. The schedules for $\tau^{(2)}$ and $\tau^{(3)}$ are too lengthy to be depicted; we verified them using two independently-coded EPDF simulators. ■

7.2 Tardiness Bounds for EPDF

In this section, we present results concerning tardiness bounds that can be guaranteed under EPDF. The sufficient condition for a tardiness of $q \geq 1$ quanta as given by Srinivasan and Anderson in [106] requires that the sum of the weights of the $M-1$ heaviest tasks be less than $\frac{qM+1}{q+1}$. This can be ensured if the weight of each task is restricted to be at most $\frac{q}{q+1}$. We next show that an improved weight restriction of $\frac{q+2}{q+3}$ per task is sufficient to guarantee a tardiness of q quanta. This restriction is stated below.

> **(W)** The weight of each task in the task system under consideration is at most $\frac{q+2}{q+3}$.

In what follows, we prove the following theorem.

Theorem 7.2 *Tardiness under* EPDF *is at most* q *quanta, where* $q \geq 1$*, for every GIS task system that is feasible on* $M \geq 3$ *processors and that satisfies* (W).

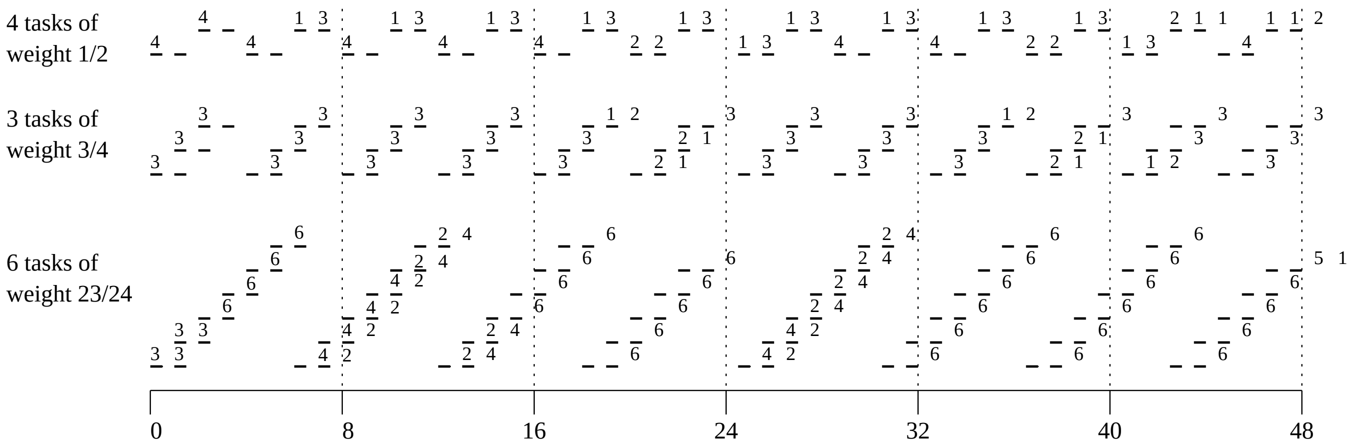


Figure 7.1: Counterexample to prove that tardiness under EPDF can exceed one quantum. 13 periodic tasks with total utilization ten are scheduled on ten processors. In the schedule, tasks of the same weight are shown together as a group. Each column corresponds to a time slot. The PF-window of each subtask is shown as a sequence of dashes that are aligned. An integer value n in slot t means that n tasks in the corresponding group have a subtask scheduled at t. Subtasks that miss deadlines are shown scheduled after their windows. Ties are broken in favor of tasks with lower weights. In this schedule, 11 subtasks miss their deadlines at time 48. Hence, tardiness is 2 quanta for at least one subtask.

To prove the above theorem, we use a setup similar to that used by Srinivasan and Anderson in [105] and [106] (which was also used in Chapter 6). (Again, it should be pointed out that the details of the current proof differ significantly from the rest.) As in Chapter 6, our proof is by contradiction. Hence, we assume that Theorem 7.2 does not hold, and as before, we define a task system that is minimal in some sense for which the theorem to be proved does not hold. Thus, for the problem at hand, our assumption implies that there exists a $q \geq 1$, and a time t_d and a concrete task system σ defined as follows.

Definition 7.1: t_d is the earliest deadline of a subtask with a tardiness of $q+1$ under EPDF in any feasible GIS task system satisfying (W), *i.e.*, there exists some such task system with a subtask with deadline at t_d and tardiness $q+1$, and there does not exist any such task system with a subtask with deadline prior to t_d and a tardiness of $q+1$.

Definition 7.2: σ is a feasible concrete GIS task system satisfying (W) with the following properties.

(**S1**) A subtask in σ with deadline at t_d has a tardiness of $q+1$ under EPDF.

(**S2**) No feasible concrete task system satisfying (W) and (S1) releases fewer subtasks in $[0, t_d)$ than σ.

	Task Set		U_{sum}	Tardiness (in quanta) on $M = U_{sum}$ processors
	# of tasks	weight		
$\tau^{(1)}$	4	1/2	10	2 at 50
	3	3/4		
	6	23/24		
$\tau^{(2)}$	4	1/2	19	3 at 963
	3	3/4		
	5	23/24		
	10	239/240		
$\tau^{(3)}$	4	1/2	80	4 at 43,204
	3	3/4		
	3	23/24		
	1	31/32		
	4	119/120		
	4	239/240		
	6	479/480		
	8	959/960		
	15	1199/1200		
	15	2399/2400		
	20	4799/4800		

Table 7.1: Counterexamples to show that tardiness under EPDF can exceed three.

In what follows, let $\hat{\mathcal{S}}$ denote an EPDF schedule for σ in which a subtask of σ with deadline at t_d has a tardiness of $q+1$.

By (S1) and (S2), exactly one subtask in σ has a tardiness of $q+1$: if several such subtasks exist, then all but one can be removed and the remaining subtask will still have a tardiness of $q+1$, contradicting (S2). Similarly, a subtask with deadline later than t_d cannot impact how subtasks with deadlines at or before t_d are scheduled. Therefore, no subtask in σ has a deadline after t_d. Based on these facts, Lemma 7.1 below can be shown to hold. First, we claim the following.

Claim 7.1 *There is no hole in any slot in* $[t_d - 1, t_d + q)$ *in* $\hat{\mathcal{S}}$.

Proof: As discussed above, by Definition 7.1, (S1), and (S2), there is exactly one subtask in σ that has a tardiness of $q+1$. Let T_i denote the only such subtask. By (S1) again, the deadline of T_i is at t_d, and hence, T_i is scheduled at time $t_d + q$.

The proof of the claim is by induction on decreasing time t. We start by showing that there is no hole in slot $t_d + q - 1$.

Base Case: $t = t_d + q - 1$. Let T_h denote the predecessor, if any, of T_i. Because the deadlines of any two successive subtasks of the same task differ by at least one time unit, $\mathsf{d}(T_h) \leq t_d - 1$ holds. Therefore, by Definition 7.1, the tardiness of T_h is at most q, and T_h completes executing by $t_d + q - 1$. Hence, no subtask of T is scheduled in slot $t_d + q - 1$. Thus, there is no hole in slot $t_d + q - 1$; otherwise, EPDF would schedule T_i there.

Induction Hypothesis: Assume that there is no hole in slots $[t', t_d + q)$, where $t_d - 1 < t' < t_d + q$.

Induction Step: $t = t' - 1$. We show that there is no hole in slot $t' - 1$. The deadline of every subtask scheduled in t' is at most t_d. Hence, the release time and the eligibility time of every such subtask is at or before $t_d - 1$. Since $t_d - 1 \leq t' - 1$, every subtask scheduled at t' can be scheduled at $t' - 1$ unless its predecessor is scheduled at $t' - 1$. By the induction hypothesis, there is no hole in slot t'. Hence, if there is a hole in $t' - 1$, then at most $M - 1$ of the M subtasks scheduled at t' can have their predecessors scheduled at $t' - 1$, implying that at least one of the subtasks scheduled at t' should have been scheduled at $t' - 1$, which is a contradiction. Therefore, there can be no hole in $t' - 1$. ■

We now show that LAG of σ at t_d is exactly $qM + 1$.

Lemma 7.1 $\mathsf{LAG}(\sigma, t_d, \hat{\mathcal{S}}) = qM + 1$.

Proof: By Claim 7.1, there is no hole in any slot in $[t_d, t_d + q)$ in $\hat{\mathcal{S}}$. Further, the subtask with a tardiness of $q + 1$ and deadline at t_d, as specified in (S1), is not scheduled until time $t_d + q$. (Also, recall that there is exactly one such subtask.) Thus, because every subtask in σ has a deadline of at most t_d, there exist exactly $qM + 1$ subtasks with deadlines at most t_d that are pending at t_d in $\hat{\mathcal{S}}$. In the ideal schedule, all of these subtasks complete executing by time t_d. Therefore, the LAG of σ at t_d, which is the difference between the ideal allocation and the allocation in $\hat{\mathcal{S}}$ in $[0, t_d)$, is $qM + 1$. ■

By Claim 7.1, there is no hole in slot $t_d - 1$. Therefore, by the contrapositive of Lemma 3.4, $\mathsf{LAG}(\sigma, t_d - 1, \hat{\mathcal{S}}) \geq \mathsf{LAG}(\sigma, t_d, \hat{\mathcal{S}})$, which, by Lemma 7.1, is $qM + 1$. Thus, because $\mathsf{LAG}(\sigma, 0, \hat{\mathcal{S}}) = 0$, there exists a time t, where $0 \leq t < t_d - 1$ such that $\mathsf{LAG}(\sigma, t, \hat{\mathcal{S}}) < qM + 1$ and

$\mathsf{LAG}(\sigma, t+1, \hat{\mathcal{S}}) \geq qM + 1$. This further implies the existence of a time $0 \leq t_h < t_d - 1$, a concrete task system τ, and an EPDF schedule $\mathcal{S}$ for τ defined as follows.

Definition 7.3: $0 \leq t_h < t_d - 1$ is the earliest time such that the LAG in any EPDF schedule for any feasible concrete GIS task system satisfying (W) is at least $qM + 1$ at $t_h + 1$.

Definition 7.4: τ is a feasible concrete GIS task system satisfying (W) with the following properties.

(**T1**) $\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) \geq qM + 1$.

(**T2**) No feasible concrete task system satisfying (W) and (T1) releases fewer subtasks than τ.

(**T3**) No feasible concrete task system satisfying (W), (T1), and (T2) has a larger rank than τ where the *rank* of a task system is the sum of the eligibility times of all its subtasks, *i.e.*, $rank(\tau, t) = \sum_{\{T_i \in \tau\}} \mathsf{e}(T_i)$.

(T2) can be thought of as identifying a minimal task system in the sense of having LAG exceed $qM + 1$ at the earliest possible time with the fewest number of subtasks, subject to satisfying (W). As already explained, if Theorem 7.2 does not hold for all task systems satisfying (W), then there exists some task system whose LAG is at least $qM + 1$. Therefore, some task system satisfying (W), (T1), and (T2) necessarily exists. (T3) further restricts the nature of such a task system by requiring subtask eligibility times to be spaced as much apart as possible.

Before continuing with the proof, we would like to point out the following as we did in Chapter 6. As mentioned earlier, the overall proof setup here is similar to that used in [105] for establishing the optimality of the PD^2 algorithm, and in Chapter 6 for determining a schedulable utilization bound for EPDF. Though the details of the proofs differ significantly, some basic properties hold for the task systems considered in each problem, such as task system τ in Definition 7.4. The proofs of some such basic properties may have to be reworded to exactly hold for the task system under consideration. However, to avoid repetition, we simply borrow such properties without proof, but explicitly state that when we have done so.

We next prove some properties about the subtasks of τ scheduled in $\mathcal{S}$.

Lemma 7.2 *Let T_i be a subtask in τ. Then, the following properties hold.*

(**a**) *If T_i is scheduled at t, then $\mathsf{e}(T_i) \geq \min(\mathsf{r}(T_i), t)$.*

(**b**) *If T_i is scheduled before t_d, then tardiness of T_i is at most q.*

Proof of part (a): The proof for this part is very similar to that of Lemma 6.8(d), but is provided to give a flavor of the changes in wording needed to fit the task system under consideration here. Suppose $\mathsf{e}(T_i)$ is not equal to $\min(\mathsf{r}(T_i), t)$. Then, by (3.12) and because T_i is scheduled at t, it is before $\min(\mathsf{r}(T_i), t)$. Hence, simply changing $\mathsf{e}(T_i)$ so that it equals $\min(\mathsf{r}(T_i), t)$ will not affect how T_i or any other subtask is scheduled. That is, the actual allocations in $\mathcal{S}$ to every task, and hence, the lag of every task, will remain unchanged. Therefore, the LAG of τ at $t_h + 1$ will still be at least $qM + 1$. However, changing the eligibility time of T_i increases the rank of the task system, and hence, (T3) is contradicted. ■

Proof of part (b). Follows from Definition 7.1. ■

In what follows, we show that if (W) is satisfied, then there does not exist a time t_h as defined in Definition 7.3, that is, we contradict its existence, and in turn prove Theorem 7.2. For this, we deduce the LAG of τ at $t_h + 1$ by determining the lags of the tasks in τ.

By the definition there of t_h in Definition 7.3, $\mathsf{LAG}(\tau, t_h + 1) > \mathsf{LAG}(\tau, t_h)$. Hence, by Lemma 3.4, the following holds.

(H) There is at least one hole in slot t_h.

7.2.1 Categorization of Subtasks

As can be seen from (3.22) and (3.15), the lag of a task T at t depends on the allocations that subtasks of T receive in each time slot until t in the ideal schedule. Hence, a tight estimate of such allocations is essential to bounding the lag of T reasonably accurately. If a subtask's index is not known, then (3.15), which can otherwise be used to compute the allocation received by any subtask in any slot *exactly*, is not of much help. Hence, in this subsection, we define terms that will help in categorizing subtasks, and then derive upper bounds for the allocations that these categories of subtasks receive in the ideal schedule.

***k-dependent* subtasks.** The subtasks of a heavy task with weight in the range $[1/2, 1)$ can be divided into "groups" based on their group deadlines in a straightforward manner: place all subtasks with identical group deadlines in the same *group* and identify the group using the smallest index of any subtask in that group. For example, in Figure 3.5, $G_1 = \{T_1, T_2\}$, $G_3 = \{T_3, T_4, T_5\}$, and $G_6 = \{T_6, T_7, T_8\}$. If there are no IS separations or GIS omissions among the subtasks of a group, then a deadline miss by q quanta for a subtask T_i will necessarily result in a deadline miss by at least q quanta for the remaining subtasks in T_i's group. Hence, a subtask T_j is dependent on all prior subtasks in its group for not missing its deadline. If T is heavy, we say that T_j is *k-dependent*, where $k \geq 0$, if T_j is the $(k+1)^{\text{st}}$ subtask in its group, computed assuming all subtasks are present (that is, as in the determination of group deadlines, even if T is GIS and some subtasks are omitted, k-dependency is determined assuming there are no omissions).

Recall that by Lemma 3.1 all subtasks of a heavy task with weight less than one are of length two or three. Further, note that each subtask in each group except possibly the first is of length two. This implies that for a periodic task the deadlines of any two successive subtasks that belong to the same group differ by exactly one. Also, each subtask in each group except possibly the final subtask has a b-bit of one. Finally, if the final subtask of a group has a b-bit of one, then the first subtask of the group that follows in of length three. These properties are summarized in the following three lemmas.

Lemma 7.3 *Let T be a heavy task with $wt(T) < 1$ and let T_i be a 0-dependent subtask of T. Then, one of the following holds*: **(i)** $i = 1$; **(ii)** $i > 1$ *and* $b(T_{i-1}) = 0$; **(iii)** $|\omega(T_i)| = 3$.

Lemma 7.4 *If T_i is a k-dependent subtask of a* ***periodic*** *task T, where $i \geq 2$ and $k \geq 1$, then $d(T_i) = d(T_{i-1}) + 1$ and $r(T_i) = d(T_{i-1}) - 1$.*

Lemma 7.5 *Let T_i, where $i > 1$, be a k-dependent subtask of T with $wt(T) < 1$. If $k \geq 1$, then $|\omega(T_i)| = 2$ and $b(T_{i-1}) = 1$.*

If a task T is light, then we simply define all of its subtasks to be 0-dependent. In this case, each subtask is in its own group.

Miss initiators. A subtask scheduled at t and missing its deadline by c quanta, where $c \geq 1$, is referred to as a *miss initiator by c* (or a c-MI, for short) for its group, if no subtask of the same task is scheduled at $t-1$. (A miss initiator by q, *i.e.*, a q-MI, will simply be referred to as an MI.) Thus, a subtask is a c-MI if it misses its deadline by c quanta and is either the first subtask in its group to do so or is separated from its predecessor by an IS or GIS separation, and its predecessor is not scheduled in the immediately preceding slot. Such a subtask is termed a miss initiator by c because in the absence of future separations, it causes all subsequent subtasks in its group to miss their deadlines by c quanta as well. Several examples of MIs for $q = 1$ are shown in Figure 7.2.

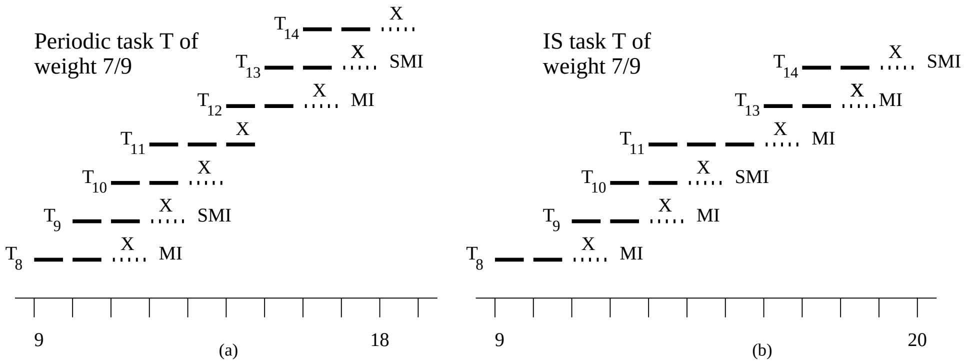


Figure 7.2: Possible schedules for the second job of **(a)** a periodic and **(b)** an IS task of weight 7/9 under EPDF. Subtasks are scheduled in the slots marked by an X. Solid (dotted) lines indicate slots that lie within (outside) the window of a subtask. A subtask scheduled in a dotted slot misses its deadline. In (a), T_8 and T_{12} are MIs, T_9 and T_{13} are SMIs, and the remaining subtasks fall within neither category. T_{10} and T_{14} have a tardiness of one, and T_{11} has a tardiness of zero. In (b), T_8, T_9, T_{11}, and T_{13} are MIs, and T_{10} and T_{14} are SMIs. Note that T_8 and T_9 (T_{11} and T_{13}) belong to the same group G_8 (G_{11}). Thus, if there are IS separations, there may be more than one MI in a group.

Successors of miss initiators. The immediate successor T_{i+1} of a c-MI T_i is called a *successor of a c-MI* (or c-SMI, for short) if $tardiness(T_{i+1}) = tardiness(T_i) = c$, $\mathcal{S}(T_{i+1}, t) = 1 \Rightarrow \mathcal{S}(T_i, t-1) = 1$, and T_i is a c-MI. (A successor of a miss initiator by q, *i.e.*, a q-SMI, will simply be referred to as an SMI.) Figure 7.2 shows several examples for $q = 1$. Note that for T_{i+1} to be a c-SMI, its predecessor in $\mathcal{S}$ must be T_i, rather than some lower-indexed subtask of T.

The lemma below follows immediately from Lemma 7.3, which implies that the deadline of the first subtask of a group differs from the final subtask of the preceding group by at least two.

Lemma 7.6 *Let T_i be a subtask that is scheduled at t and let T_i's tardiness be $c > 0$ quanta. If T_j, where $j < i$, is scheduled at $t-1$ and T_j does not belong to the same group as T_i, then the tardiness of T_j is at least $c+1$.*

The three lemmas that follow bound the allocation received by a k-dependent subtask in the first slot of its window in the ideal schedule.

Lemma 7.7 *The allocation $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i))$ received in the ideal schedule by a k-dependent subtask T_i of a **periodic** task T with $wt(T) < 1$ in the first slot of its window is at most $k \cdot \frac{T.e}{T.p} - (k-1) - \frac{1}{T.p}$, for all $k \geq 0$.*

Proof: The proof is by induction on k.

Base Case: $k = 0$. Because $wt(T) < 1$, and $T.e$ and $T.p$ are integral, $T.e \leq T.p - 1$. Thus, by (3.18), $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) \leq wt(T) \leq 1 - 1/T.p$, and the lemma holds for the base case.

Induction Step: Assuming that the lemma holds for $(k-1)$-dependent subtasks, we show that it holds for k-dependent subtasks, where $k \geq 1$. Because $k \geq 1$, by the definition of k-dependency, $i > 1$ and T is heavy. Hence, by Lemma 3.1, $|\omega(T_{i-1})|$ is either two or three. We consider two cases.

Case 1: $|\omega(T_{i-1})| = 2$. Since $k \geq 1$, T_{i-1} is $(k-1)$-dependent. Therefore, by the induction hypothesis,

$$\mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{r}(T_{i-1})) \leq (k-1) \cdot (T.e/T.p) - (k-2) - (1/T.p). \tag{7.1}$$

Because $|\omega(T_{i-1})| = 2$, by (3.17), $\mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1) = 1 - \mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{r}(T_{i-1}))$. Hence, by (7.1), $\mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1) \geq (k-1) + (1/T.p) - (k-1) \cdot (T.e/T.p)$. Because T_i is k-dependent, where $k \geq 1$, by Lemma 7.5, $b(T_{i-1}) = 1$, and by Lemma 3.3, $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) = (T.e/T.p) - \mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1) \leq k \cdot (T.e/T.p) - (k-1) - (1/T.p)$.

Case 2: $|\omega(T_{i-1})| = 3$. By Lemma 7.5, T_{i-1} is 0-dependent; hence, T_i is 1-dependent, *i.e.*, $k = 1$. By Lemma 7.5, $\mathsf{b}(T_{i-1}) = 1$, and hence, by Lemma 3.3,

$$\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) = \frac{T.e}{T.p} - \mathsf{A}(\mathsf{PS}, T_i, \mathsf{d}(T_{i-1}) - 1) \leq \frac{T.e}{T.p} - \frac{1}{T.p} \qquad \text{(by 3.19).} \qquad \blacksquare$$

Lemma 7.8 *The allocation $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i))$ received in the ideal schedule by a k-dependent subtask T_i of a GIS task T in the first slot of its window is at most $k \cdot \frac{T.e}{T.p} - (k-1) - \frac{1}{T.p}$.*

Proof: Follows from Lemma 7.7 and the definition of GIS tasks. (The allocation that T_i receives in each slot of its window is identical to the allocation that it would receive if T were periodic.) $\blacksquare$

Lemma 7.9 *Let T_i, where $i \geq k+1$ and $k \geq 1$, be a subtask of T with $wt(T) < 1$ such that $|\omega(T_i)| \geq 3$ and $b(T_{i-1}) = 1$. Let the number of subtasks in T_{i-1}'s dependency group be at least k. Then, $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) \leq k \cdot \frac{T.e}{T.p} - (k-1) - \frac{1}{T.p}$.*

Proof: Since $|\omega(T_i)| \geq 3$, by Lemma 7.5, T_i is 0-dependent and is the first subtask in its group. Hence, T_{i-1} is the final subtask in its dependency group, and since there are at least k subtasks in T_{i-1}'s group, T_{i-1} is $(k-1)$-dependent. Hence, by Lemma 7.8, $\mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{r}(T_{i-1})) \leq (k-1) \cdot \frac{T.e}{T.p} - (k-2) - \frac{1}{T.p}$. (What follows is similar to the reasoning used in the induction step in Lemma 7.7.) If $|\omega(T_{i-1})| = 2$, then, by (3.17), $\mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1})) \geq 1 - ((k-1) \cdot \frac{T.e}{T.p} - (k-2) - \frac{1}{T.p}) = (k-1) - (k-1) \cdot \frac{T.e}{T.p} + \frac{1}{T.p}$. By the statement of the lemma, $b(T_{i-1}) = 1$, and hence, by Lemma 3.3, $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) = wt(T) - \mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1) \leq k \cdot \frac{T.e}{T.p} - (k-1) - \frac{1}{T.p}$. Thus, the lemma holds when $|\omega(T_{i-1})| = 2$.

On the other hand, if $|\omega(T_{i-1})| \geq 3$, then by Lemma 7.5, T_{i-1} is 0-dependent. By (3.19), $\mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1) \geq \frac{1}{T.p}$, and hence, because $b(T_{i-1}) = 1$, by Lemma 3.3, $\mathsf{A}(\mathsf{PS}, T_i, \mathsf{r}(T_i)) = wt(T) - \mathsf{A}(\mathsf{PS}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1) \leq \frac{T.e}{T.p} - \frac{1}{T.p}$. By the statement of the lemma, $|\omega(T_i)| = 3$, and hence, T_i is also 0-dependent. Thus, T_{i-1} is the only subtask in its group, and hence, $k = 1$. (Note that k here denotes the number of subtasks that are in the same dependency group as T_{i-1}.) Therefore, the lemma holds for this case too. ■

Having determined a bound on the allocation received by a subtask in the first slot of its window in the ideal schedule based on its k-dependency, we next bound the lag of a task at time t, this time based on the k-dependency of its last-scheduled subtask.

Lemma 7.10 *Let T_i be a k-dependent subtask of a task T for $k \geq 0$, and let the tardiness of T_i be s for some $s \geq 1$ (that is, T_i is scheduled at time $\mathsf{d}(T_i) + s - 1$). Then $\mathsf{lag}(T, \mathsf{d}(T_i) + s) < (k + s + 1) \cdot wt(T) - k$.*

Proof: By the statement of the lemma, T_i and all prior subtasks of T are scheduled in $[0, \mathsf{d}(T_i) + s)$. Hence, $\mathsf{lag}(T, \mathsf{d}(T_i) + s)$ depends on the number of subtasks of T after T_i released prior to $\mathsf{d}(T_i) + s$, the allocations they receive in the ideal schedule, and when they are scheduled in $\mathcal{S}$. It can be verified from (3.9), (3.10), and (3.8) that at most $s + 1$ successors of T_i — $T_{i+1}, \ldots, T_{i+s+1}$ — are released before $\mathsf{d}(T_i) + s$. Hence, the lag of T at $\mathsf{d}(T_i) + s$ in $\mathcal{S}$ is maximized if all those subtasks are present and are released without any IS separations and $\mathcal{S}$ has not scheduled any of them by time $\mathsf{d}(T_i) + s$. We will assume that this is the case. By Lemma 6.10, at most one successor of T_i, namely T_{i+1}, can have a release time that is before $\mathsf{d}(T_i)$. Further, $\mathsf{r}(T_{i+1}) \geq \mathsf{d}(T_i) - 1$ holds. Hence, $\mathsf{lag}(T, \mathsf{d}(T_i) + s) \leq \mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{d}(T_i) - 1)) + \mathsf{A}(\mathsf{PS}, T, \mathsf{d}(T_i), \mathsf{d}(T_i) + s)$. If $\mathsf{r}(T_{i+1}) > \mathsf{d}(T_i) - 1$ holds, then $\mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{d}(T_i) - 1) = 0$. On the other hand, if $\mathsf{r}(T_{i+1}) = \mathsf{d}(T_i) - 1$, then by (3.6), $\mathsf{b}(T_i) = 1$. Further, either $|\omega(T_{i+1})| = 2$ or $|\omega(T_{i+1})| > 2$. In the former case, T is heavy, and by the definition of k-dependency, T_{i+1} belongs to the same dependency group as T_i and is $(k+1)$-dependent. Hence, by Lemma 7.8, $\mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{r}(T_{i+1})) \leq (k+1) \cdot wt(T) - k - \frac{1}{T.p}$. If the latter holds, *i.e.*, $|\omega(T_{i+1})| > 2$, we reason as follows. Since T_i is k-dependent, the number of subtasks in T_i's group is at least $k+1$. Therefore, since $\mathsf{b}(T_i) = 1$, Lemma 7.9 applies for T_{i+1} and it follows that $\mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{r}(T_{i+1})) \leq (k+1) \cdot wt(T) - k - \frac{1}{T.p}$. Thus, in either case, $\mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{d}(T_i) - 1) = \mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{r}(T_{i+1})) \leq (k+1) \cdot wt(T) - k - \frac{1}{T.p}$.

By (3.16), $\mathsf{A}(\mathsf{PS}, T, \mathsf{d}(T_i), \mathsf{d}(T_i)+s) \leq s \cdot wt(T)$. Hence, $\mathsf{lag}(T, \mathsf{d}(T_i)+s) \leq \mathsf{A}(\mathsf{PS}, T_{i+1}, \mathsf{d}(T_i) - 1)) + \mathsf{A}(\mathsf{PS}, T, \mathsf{d}(T_i), \mathsf{d}(T_i)+s) \leq (k+s+1) \cdot wt(T) - k - \frac{1}{T.p} < (k+s+1) \cdot wt(T) - k$. ■

7.2.2 Subclassification of Tasks in $A(t)$

Recall from Section 3.6 that a task in $A(t)$ is scheduled in slot t. We further classify tasks in $A(t)$, based on the tardiness of their subtasks scheduled at t, as follows.

$A_0(t)$: Includes T in $A(t)$ iff its subtask scheduled at t has zero tardiness.

$A_q(t)$: Includes T in $A(t)$ iff its subtask scheduled at t has a tardiness of q.

$A_{q-1}(t), q > 1$: Includes T in $A(t)$ iff its subtask scheduled at t has a tardiness greater than 0 and less than q.

$A_q(t)$ is further partitioned into $A_q^0(t)$, $A_q^1(t)$, and $A_q^2(t)$.

$A_q^0(t)$: Includes T in $A_q(t)$ iff its subtask scheduled at t is an MI.

$A_q^1(t)$: Includes T in $A_q(t)$ iff its subtask scheduled at t is an SMI.

$A_q^2(t)$: Includes T in $A_q(t)$ iff its subtask scheduled at t is neither an MI nor an SMI.

$A_{q-1}^0(t), q > 1$: Includes T in $A_{q-1}(t)$ iff its subtask scheduled at t is a c-MI, where $0 < c < q$.

From the above, we have the following.

$$A_0(t) \cup A_q(t) \cup A_{q-1}(t) = A(t) \text{ and } A_q^0(t) \cup A_q^1(t) \cup A_q^2(t) = A_q(t) \tag{7.2}$$

$$A_0(t) \cap A_q(t) = A_q(t) \cap A_{q-1}(t) = A_{q-1}(t) \cap A_0(t) = \emptyset \tag{7.3}$$

$$A_q^0(t) \cap A_q^1(t) = A_q^1(t) \cap A_q^2(t) = A_q^2(t) \cap A_q^0(t) = \emptyset \tag{7.4}$$

7.2.3 Task Lags by Task Classes and Subclasses

The next eight lemmas give bounds on the lags of tasks in $A(t)$, $B(t)$, and $I(t)$ at $t+1$, where $t \leq t_h$ is a slot with a hole, and hence, hold for t_h, as well. As mentioned earlier, some of these lemmas are proved for a slightly different task system, but hold for τ as defined here in Definition 7.4 as well.

Let $t \leq t_h$ be a slot with a hole. Then, Lemmas 7.11–7.18 hold.

Lemma 7.11 (from [106]) *For* $T \in I(t), \mathsf{lag}(I, t+1) = 0$.

Lemma 7.12 (*from Chapter 6*) *For* $T \in B(t), \mathsf{lag}(B, t+1) = 0$.

Lemma 7.13 (*from Chapter 6*) *For* $T \in A_0(t), 0 \leq \mathsf{lag}(T, t+1) < wt(T)$.

Lemma 7.14 *For* $T \in A_q^0(t)$, $\mathsf{lag}(T, t+1) < (q+1) \cdot wt(T)$.

Proof: If $T \in A_q^0(t)$, then the subtask T_i of T scheduled at t is an MI, and $d(T_i) = t - q + 1$. Further T_i is k-dependent, where $k \geq 0$. Hence, by Lemma 7.10, $\mathsf{lag}(T, t+1)$ is less than $((k+q+1) \cdot wt(T) - k)$, which (because $wt(T) \leq 1$) is at most $(q+1) \cdot wt(T)$, for all $k \geq 0$. ■

The two lemmas below follow similarly.

Lemma 7.15 *For* $T \in A_q^1(t)$, $\mathsf{lag}(T, t+1) < (q+2) \cdot wt(T) - 1$.

Proof: If $T \in A_q^1(t)$, then the subtask T_i of T scheduled at t is an SMI, and is k-dependent for $k \geq 1$. Also, $d(T_i) = t - q + 1$. Thus, by Lemma 7.10, $\mathsf{lag}(T, t+1) < (k+q+1) \cdot wt(T) - k \leq (q+2) \cdot wt(T) - 1$ for all $k \geq 1$ (because $wt(T) \leq 1$). ■

Lemma 7.16 *For* $T \in A_q^2(t)$, $\mathsf{lag}(T, t+1) < (q+3) \cdot wt(T) - 2$.

Proof: Similar to that of Lemma 7.15. ■

Lemma 7.17 *For* $T \in A_{q-1}(t)$, $\mathsf{lag}(T, t+1) < q \cdot wt(T)$.

Proof: Let T_i be T's subtask scheduled at t and let s denote the tardiness of T_i. Then, $t + 1 = \mathsf{d}(T_i) + s$. Let T_i be k-dependent, where $k \geq 0$. By the definition of A_{q-1}, $0 < s < q$ holds, and by Lemma 7.10, $\mathsf{lag}(T, \mathsf{d}(T_i) + s) = \mathsf{lag}(T, t+1) < (k+s+1) \cdot wt(T) - k \leq (k+q) \cdot wt(T) - k \leq q \cdot wt(T)$, for all $k \geq 0$. ■

Lemma 7.18 *For* $T \in A(t)$, $\mathsf{lag}(T, t+1) \geq 0$.

Proof: If T is in $A_0(t)$, then the lemma follows from Lemma 7.13. Otherwise, T's subtask scheduled at t has non-zero tardiness. Hence, T could not have received more allocation in $\mathcal{S}$ than in the ideal schedule. ■

7.2.4 Some Auxiliary Lemmas

In proving Theorem 7.2, we make use of Lemmas 6.3, 6.4, and 6.5 established in prior work by Srinivasan and Anderson, which were stated in Chapter 6. (Lemmas 6.3 and 6.4 can easily be seen to hold even if τ is as defined here. Lemma 6.5 holds because, as in Chapter 7, it can be shown that if $\mathsf{d}(U_j) > t+1$ holds, then U_j has no impact on how subtasks are scheduled after t. In particular, it can be shown that even if U_j is removed, no subtask scheduled after t can be scheduled at or before t. Therefore, it can be shown that if the lemma does not hold, then the GIS task system obtained from τ by removing U_j also has a LAG at least $qM+1$ at t_h+1, which is a contradiction to (T2).)
The following lemma is analogous to Lemma 6.6 in Chapter 6. The difference is due to the presence of subtasks with non-zero tardiness in τ here. Arguments similar to those used in proving the above lemma can be used to show the following.

Lemma 7.19 (from Chapter 6) *Let $t < t_d$ be a slot with holes. Let U_j be a subtask that is scheduled at t and let the tardiness of U_j be zero. Then $\mathsf{d}(U_j) = t+1$.*

By Definition 7.3, $\mathsf{LAG}(\tau, t_h+1) > \mathsf{LAG}(\tau, t_h)$. Therefore, by Lemma 6.3, $B(t_h) \neq \emptyset$. By (H), there is at least one hole in t_h. Hence, by Lemma 6.4, the critical subtask at t_h of every task in $B(t_h)$ is scheduled before t_h. The next definition identifies the latest time at which a critical subtask at t_h of any task in $B(t_h)$ is scheduled.

Definition 7.5: t_b denotes the latest time before t_h at which any subtask of any task in $B(t_h)$ that is critical at t_h is scheduled.

U and U_j are henceforth to be taken as defined below.

Definition 7.6: U denotes a task in $B(t_h)$ with a subtask U_j scheduled at t_b that is critical at t_h.

The lemma below shows that the deadline of the critical subtask at t_h of every task in $B(t_h)$ is at t_h+1. (Though the next two lemmas have been proved with respect to t_h, they hold for any slot t with a hole (that is at or before the earliest time at which the theorem to be proved is violated) across which LAG increases in a schedule for a task system that is set up in a manner similar to that of σ or τ. In such a case, t_b and U_j should taken to be with respect to t.)

Lemma 7.20 *Let T be a task in $B(t_h)$ and let T_i be T's critical subtask at t_h. Then, $\mathsf{d}(T_i) =$*

$t_h + 1$.

Proof: Because T is in $B(t_h)$, T is active at t_h, but is not scheduled at t_h. Hence, T_i, which is critical at t_h, should have been scheduled earlier. In this case, by Lemma 6.5, $\mathsf{d}(T_i) \leq t_h + 1$ holds. However, since T_i is T's critical subtask at t_h, by Definition 3.2, $\mathsf{d}(T_i) \geq t_h + 1$ holds. Therefore, $\mathsf{d}(T_i) = t_h + 1$ follows. ■

The following lemma shows that at least one subtask scheduled in t_h has a tardiness of zero, *i.e.*, $|A_0(t_h)| \geq 1$.

Lemma 7.21 *There exists a subtask W_ℓ scheduled at t_h with $\mathsf{e}(W_\ell) \leq t_b$, $\mathsf{d}(W_\ell) = t_h + 1$, and $\mathcal{S}(W, t) = 0$, for all $t \in [t_b, t_h)$. Also, there is no hole in any slot in $[t_b, t_h)$. (Note that, by this lemma, $A_0(t_h) \neq \emptyset$.)*

Proof:

We first show that the first subtask to be displaced upon U_j's removal (where U_j is as defined in Def. 7.6) has properties as stated for W_ℓ, *i.e.*, is eligible at or before t_b and has its deadline at $t_h + 1$.

Let τ' be the task system obtained by removing U_j from τ, and let $\mathcal{S}'$ be the EPDF schedule for τ'. Let $\Delta_1 = \langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ be the first valid displacement, if any, that results due to the removal of U_j. Then, $X^{(1)} = U_j$, $t_1 = t_b$, and by Lemma A.1,

$$t_2 > t_1 = t_b. \tag{7.5}$$

Let $T_i = X^{(2)}$. We first show that $t_2 \geq t_h$.

Assume to the contrary that $t_2 < t_h$. Then, by (7.5) and Definition 7.5, T is not in $B(t_h)$. Therefore, T is in $I(t_h)$ or in $A(t_h)$. In either case,

$$\mathsf{d}(T_i) \leq t_h. \tag{7.6}$$

To see this, note that if $T \in I(t_h)$, then because T is not active at t_h, by Definition 3.1, $\mathsf{d}(T_i) \leq t_h$. On the other hand, if $T \in A(t_h)$, then consider T's subtask, say T_k, scheduled at t_h. By Lemma 7.19, $\mathsf{d}(T_k) \leq t_h + 1$. Because T_i is scheduled at $t_2 < t_h$, T_i is an earlier subtask of T than T_k, and hence, by (3.10) and (3.8), $\mathsf{d}(T_i) \leq t_h$. Because is U_j is U's critical subtask at t_h and U is in $B(t_h)$, by Lemma 7.20, we have

$$\mathsf{d}(U_j) = t_h + 1. \tag{7.7}$$

By (7.6) and (7.7), $\mathsf{d}(U_j) > \mathsf{d}(T_i)$. However, since EPDF selects U_j over T_i at time t_b (which follows because the displacement under consideration is valid), this is a contradiction. Thus, our assumption that $t_2 < t_h$ holds is false.

Having shown that $t_2 \geq t_h$, we next show $t_2 = t_h$. Assume, to the contrary, that $t_2 > t_h$. Since displacement $\Delta_1 = \langle U_j, t_b, T_i, t_2 \rangle$is valid, $\mathsf{e}(T_i) \leq t_b$. This implies that T_i is eligible at t_h, and because there is a hole in t_h, it should have been scheduled there in $\mathcal{S}$, and not later at t_2. It follows that $t_2 = t_h$.

Finally, because U_j is scheduled at t_b in preference to T_i, $\mathsf{d}(T_i) \geq \mathsf{d}(U_j) = t_h + 1$ (from (7.7)), which by Lemma 6.5 (since T_i is scheduled in slot t_h) implies that

$$\mathsf{d}(T_i) = t_h + 1. \tag{7.8}$$

Thus, the first subtask, if any, to be displaced upon U_j's removal satisfies the properties specified for W_ℓ in the statement of the lemma. Hence, if a subtask with such properties does not exceed, then U_j's removal will not lead to any displacements.

Next, we show that unless the other two conditions specified in the lemma also hold, no subtask will be displaced upon U_j's removal. For this, first note that by (7.7) and (7.8) T_i and U_j have equal deadlines, and hence, T_i is not U_j's successor. Next, note that because $\langle U_j, t_b, T_i, t_h \rangle$ is valid, no subtask of T prior to T_i is scheduled in $[t_b, t_h)$, and also if there is a hole in any slot t in $[t_b, t_h)$, then EPDF would have scheduled T_i at t.

Thus, if the lemma is false, then removing U_j does not result in any displacements. We now show that, in such a case, $\mathsf{LAG}(\tau', t_h+1, \mathcal{S}') \geq qM+1$. $\mathsf{LAG}(\tau', t_h+1, \mathcal{S}') = \mathsf{A}(\mathsf{PS}, \tau', 0, t_h+1) - \mathsf{A}(\mathcal{S}', \tau', 0, t_h+1)$. τ' contains every subtask that is in τ except U_j. U_j is scheduled before t_h in $\mathcal{S}$, and by (7.7), $\mathsf{d}(U_j) = t_h + 1$. Therefore, U_j receives an allocation of one quantum by time t_h+1 in the ideal schedule for τ, and hence, $\mathsf{A}(\mathsf{PS}, \tau', 0, t_h+1) = \mathsf{A}(\mathsf{PS}, \tau, 0, t_h+1) - 1$. Similarly, since no subtask other than U_j of τ is displaced or removed in $\mathcal{S}'$, the total allocation to τ' in $\mathcal{S}'$ up to time t_h+1, $\mathsf{A}(\mathcal{S}', \tau', 0, t_h+1)$, is $\mathsf{A}(\mathcal{S}, \tau, 0, t_h+1) - 1$. Therefore, $\mathsf{LAG}(\tau', t_h+1, \mathcal{S}') = \mathsf{A}(\mathsf{PS}, \tau, 0, t_h+1) - \mathsf{A}(\mathcal{S}, \tau, 0, t_h+1) = \mathsf{LAG}(\tau, t_h+1, \mathcal{S}) \geq qM+1$ (by (T1)). To conclude, we have shown that, τ' with one fewer subtask than τ also has a LAG of at least $qM+1$ at t_h+1, which contradicts (T2). ■

Lemma 7.22 *Let $t_m \leq t_h + 1$ be a slot in which a c-MI is scheduled, where $c \geq 1$. Then, the following hold.*

(a) *For all t, where $t_m - (c+2) < t < t_m$, there is no hole in slot t, and for each subtask V_k that is scheduled in t, $\mathsf{d}(V_k) \le t_m - c + 1$.*

(b) *Let W be a task in $B(t_m)$ and let the critical subtask W_ℓ of W at t_m be scheduled before t_m. Then, W_ℓ is scheduled at or before $t_m - (c+2)$.*

Proof of part (a). The proof is by induction on decreasing t. We start with $t = t_m - 1$.

Base Case: $t = t_m - 1$. Let T_i be an c-MI scheduled at t_m. (By the statement of the lemma, at least one c-MI is scheduled in t_m.) Then, $\mathsf{d}(T_i) = t_m - c + 1$, and $\mathcal{S}(T, t_m - 1) = 0$, from the definition of an c-MI. Hence, T_i is eligible at $t_m - 1$. Because T_i is not scheduled at $t_m - 1$, it follows that there is no hole in $t_m - 1$ and that the priority of every subtask V_k scheduled at $t_m - 1$ is at least that of T_i, *i.e.*, $\mathsf{d}(V_k) \le \mathsf{d}(T_i) = t_m - c + 1$.

Induction Hypothesis: Assume that the claim holds for all t, where $t' + 1 \le t \le t_m - 1$ and $t_m - (c+1) < t' + 1 < t_m$.

Induction Step: We now show that the claim holds for $t = t'$. By the induction hypothesis, there is no hole in $t' + 1$ and $\mathsf{d}(T_i) \le t_m - c + 1$ holds for every subtask T_i scheduled in $t' + 1$. Therefore, since $wt(T) < 1$, by (3.9), $\mathsf{r}(T_i) \le t_m - c - 1$. Thus, there are M subtasks with a release time at or before $t_m - c - 1$ and deadline at or before $t_m - c + 1$ scheduled at $t' + 1 \ge t_m - c$. If there is either a hole in t' or a subtask with deadline later than $t_m - c + 1$ is scheduled in t', then there is at least one subtask scheduled in $t' + 1$ whose predecessor is not scheduled in t'. Such a subtask is eligible at t', since its release time is at or before $t_m - c - 1 \le t'$. Hence, if there is a hole in t', then the work-conserving behavior of EPDF is contradicted. Otherwise, the pseudo-deadline-based scheduling of EPDF is contradicted. Hence, the claim holds for $t = t'$. ■

Proof of part (b). By Definition 3.2, $\mathsf{d}(W_\ell) \ge t_m + 1$. Hence, since $q \ge 1$, this part easily follows from part (a). ■

7.2.5 Core of the Proof

Having classified the tasks at t_h and determined their lags at t_h, we next show that if (W) holds, then $\mathsf{LAG}(\tau, t_h + 1) < M + 1$ in each of the following cases.

For conciseness, in what follows, we denote subsets $A(t_h)$, $B(t_h)$, and $I(t_h)$ as A, B, and I, respectively. Subsets $A_{q-1}(t_h)$ and $A_q(t_h)$ and their subsets are similarly denoted without the time parameter.

Case A: $A_q = \emptyset$.

Case B: $A_q^0 \neq \emptyset$ or ($A_q^1 \neq \emptyset$ and $A_{q-1}^0 \neq \emptyset$).

Case C: $A_q^0 = \emptyset$ and $A_q^1 \neq \emptyset$ and $A_{q-1}^0 = \emptyset$.

Case D: $A_q^0 = A_q^1 = \emptyset$.

To see that the above cases are exhaustive, note that because $t_h < t_d$, by Lemma 7.2(b), the tardiness of a subtask scheduled at $t_h - 1$ is at most q. Hence, Lemma 7.6 implies that if subtask T_i scheduled at t_h has a tardiness of q, then a prior subtask of T that does not belong to the same k-dependent group as T_i cannot be scheduled at $t_h - 1$; that is, T_i is either an MI, or T_{i-1} scheduled at $t_h - 1$, and T_{i-1} and T_i belong to the same k-dependent group.

The following notation is used to denote subset cardinality.

$$a_0 = |A_0|;\ a_q^0 = |A_q^0|;\ a_q = |A_q|;\ a_q^1 = |A_q^1|;\ a_q^2 = |A_q^2|;\ a_{q-1}^0 = |A_{q-1}^0|;\ a_{q-1} = |A_{q-1}|.$$

h is defined as follows.[2]

$$h \stackrel{\text{def}}{=} \text{number of holes in } t_h$$

Because there is at least one hole in t_h

$$h > 0. \tag{7.9}$$

In the remainder of this chapter, let $W_{\max}$ denote the maximum weight of any task in τ. That is,

$$W_{\max} = \max_{T \in \tau}\{wt(T)\}. \tag{7.10}$$

In each of the above cases, $\mathsf{LAG}(\tau, t_h + 1)$ can be expressed as follows.

$$\begin{aligned} &\mathsf{LAG}(\tau, t_h + 1) \\ &= \sum_{T \in \tau} \mathsf{lag}(T, t_h + 1) \end{aligned}$$

[2]There is no correspondence between h as defined here and the subscript h in t_h. The subscript h in t_h is just an indication that t_h is a slot with holes.

$$
\begin{aligned}
\leq\ & \sum_{T \in A_0} \mathsf{lag}(T, t_h + 1) + \sum_{T \in A_{q-1}} \mathsf{lag}(T, t_h + 1) + \sum_{T \in A_q^0} \mathsf{lag}(T, t_h + 1) + \sum_{T \in A_q^1} \mathsf{lag}(T, t_h + 1) \\
& + \sum_{T \in A_q^2} \mathsf{lag}(T, t_h + 1) \qquad\qquad \text{(by (3.33), (7.2), and Lemmas 7.11 and 7.12)} \\
<\ & \sum_{T \in A_0} wt(T) + \sum_{T \in A_{q-1}} q \cdot wt(T) + \sum_{T \in A_q^0} (q+1) \cdot wt(T) + \sum_{T \in A_q^1} ((q+2) \cdot wt(T) - 1) \\
& + \sum_{T \in A_q^2} ((q+3) \cdot wt(T) - 2) \qquad\qquad \text{(from Lemmas 7.13 – 7.17)}
\end{aligned}
$$

Using (7.10), $\mathsf{LAG}(\tau, t_h + 1)$ can be bounded as

$$
\begin{aligned}
& \mathsf{LAG}(\tau, t_h + 1) \\
& \quad < \ a_0 \cdot W_{\max} + a_{q-1} \cdot q \cdot W_{\max} + a_q^0 \cdot (q+1) \cdot W_{\max} \\
& \quad \qquad + a_q^1 \cdot ((q+2) \cdot W_{\max} - 1) + a_q^2 \cdot ((q+3) \cdot W_{\max} - 2) \qquad (7.11) \\
& \quad \leq \ \begin{cases} a_0 \cdot W_{\max} + a_q^0 (q+1) W_{\max} + a_q^1 ((q+2) W_{\max} - 1) + \\ (a_{q-1} + a_q^2)((q+3) W_{\max} - 2), & W_{\max} \geq \frac{2}{3} \\ a_0 \cdot (2/3) + a_q^0 (q+1)(2/3) + a_q^1 ((q+2)(2/3) - 1) + \\ (a_{q-1} + a_q^2)(q \cdot (2/3)), & W_{\max} < \frac{2}{3} \end{cases} \qquad (7.12) \\
& \qquad\qquad \text{(because } (q+3) W_{\max} - 2 \geq q \cdot W_{\max} \text{ for } W_{\max} \geq 2/3\text{).}
\end{aligned}
$$

Note that though $(q+3)W_{\max} - 2 < q \cdot W_{\max}$ holds, for $W_{\max} < 2/3$, $(q+3) \cdot (2/3) - 2 = (2/3) \cdot q > q \cdot W_{\max}$ holds for all $W_{\max} < 2/3$. Therefore, if the values of a_0, a_{q-1}, and a_q^i are not dependent on whether $W_{\max} \geq 2/3$ or $W_{\max} < 2/3$, determining a bound on $\mathsf{LAG}(\tau, t_h + 1)$ using the expression corresponding to $W_{\max} \geq 2/3$ in (7.12) (assuming that $W_{\max} \geq 2/3$) serves as an upper bound for LAG when $W_{\max} < 2/3$. Hence, later in the chapter, when a_0, a_{q-1}, and a_q^i are not dependent on $W_{\max}$, we bound $\mathsf{LAG}(\tau, t_h + 1)$ in this way.

The total number of processors, M, expressed in terms of the number of subtasks in each subset of A scheduled at t_h, and the number of holes in t_h, is as follows.

$$
M = a_0 + a_{q-1} + a_q^0 + a_q^1 + a_q^2 + h \qquad (7.13)
$$

7.2.5.1 Case A: $A_q = \emptyset$

Case A is dealt with as follows.

Lemma 7.23 *If $A_q = \emptyset$, then* $\mathsf{LAG}(\tau, t_h + 1) < qM + 1$.

Proof: If $A_q = \emptyset$, then

$$\begin{aligned}
\mathsf{LAG}(\tau, t_h+1) &< a_0 \cdot W_{\max} + a_{q-1} \cdot q \cdot W_{\max} && \text{(by (7.11) and } a_q^0 = a_q^1 = a_q^2 = 0) \\
&\leq a_0 \cdot q \cdot W_{\max} + a_{q-1} \cdot q \cdot W_{\max} \\
&< (M-h) \cdot q \cdot W_{\max} && \text{(by (7.13), } a_0 + a_{q-1} = M - h) \\
&< qM + 1.
\end{aligned}$$ ■

Hence, if no subtask with a tardiness of q is scheduled in t_h, then (T1) is contradicted.

7.2.5.2 Case B: $A_q^0 \neq \emptyset$ or ($A_q^1 \neq \emptyset$ and $A_{q-1}^0 \neq \emptyset$)

By Lemma 7.14, $\mathsf{lag}(T, t_h+1)$ could be as high as $(q+1)\cdot wt(T)$, if the subtask T_i of T scheduled at t_h is an MI , *i.e.,* is in A_q^0. Therefore, if a_q^0 is large, then LAG could exceed $qM+1$. However, as we show below, if $a_q^0 \geq 2h-2$, then $\mathsf{LAG}(\tau, t_h+1) \leq \mathsf{LAG}(\tau, t_h) < qM+1$, contradicting (T1).

We begin by giving a lemma concerning the sum of the weights of tasks in I.

Lemma 7.24 *If* $\mathsf{LAG}(\tau, t_h+1) > \mathsf{LAG}(\tau, t_h)$, *then* $\sum_{V \in I} wt(V) < h$.

Proof: By (3.27),

$$\begin{aligned}
\mathsf{LAG}(\tau, t_h+1) &= \mathsf{LAG}(\tau, t_h) + \sum_{T \in \tau} (\mathsf{A}(\mathsf{PS}, T, t_h) - \mathcal{S}(T, t_h)) \\
&= \mathsf{LAG}(\tau, t_h) + \sum_{T \in A \cup B} (\mathsf{A}(\mathsf{PS}, T, t_h)) - (M-h) \\
&\qquad \text{(by (3.33) and } \mathsf{A}(\mathsf{PS}, T, t_h) = 0 \text{ for } T \text{ in } I \text{, and (7.13))} \\
&\leq \mathsf{LAG}(\tau, t_h) + \sum_{T \in A \cup B} wt(T) - (M-h) \qquad \text{(by (3.16))}.
\end{aligned}$$

If $\mathsf{LAG}(\tau, t_h+1) > \mathsf{LAG}(\tau, t_h)$, then by the derivation above,

$$\sum_{T \in A \cup B} wt(T) > M - h. \tag{7.14}$$

By (3.32) and (3.33), $\sum_{T \in I} wt(T) \leq M - \sum_{T \in A \cup B} wt(T)$, which by (7.14) implies that $\sum_{T \in I} wt(T) < h$. ■

We next determine the largest number of MIs and SMIs that may be scheduled at t_h, for $\sum_{T \in I} wt(T) < h$ to hold. We begin with a lemma that gives the latest time that a subtask of a task in B may be scheduled if $a_q^0 > 0$ or ($a_q^1 > 0$ and $a_{q-1}^0 > 0$).

Lemma 7.25 *If $a_q^0 > 0$ (that is, an MI is scheduled at t_h), or ($a_q^1 > 0$ and $a_{q-1}^0 > 0$) (that is, an SMI, and a c-MI, where $0 < c < q$, is scheduled at t_h), then subtask U_j defined by Definition 7.6 is scheduled no later than $t_h - (q+2)$,* i.e., *$t_b \leq t_h - (q+2)$.*

Proof: If $a_q^0 > 0$ holds, then this lemma is immediate from Definitions 7.6, 7.5, and Lemma 7.22(b). (Note that Definitions 7.6 and 7.5 imply that U_j is scheduled before t_h.)

If $a_{q-1}^0 > 0$ holds, then a c-MI, where $0 < c < q$, say T_i, is scheduled at t_h. Hence, $\mathsf{d}(T_i) = t_h + 1 - c \leq t_h$ holds. By the definition of c-MI, the predecessor of T_i is not scheduled at $t_h - 1$. Hence, the deadline of every subtask scheduled at $t_h - 1$ is at most t_h. By Definition 3.2, $\mathsf{d}(U_j) \geq t_h + 1$. Therefore, U_j, is not scheduled at $t_h - 1$.

If $a_q^1 > 0$ holds, then an SMI is scheduled at t_h, and its predecessor, which is an MI, is scheduled at $t_h - 1$. Therefore, by Lemma 7.22(b), U_j is not scheduled in $[t_h - 1 - (q+1), t_h - 1) = [t_h - (q+2), t_h - 1)$.

Thus, if both $a_{q-1}^0 > 0$ and $a_q^1 > 0$ hold, U_j is not scheduled later than $t_h - (q+3)$. ■

The lemma that follows is used to identify tasks that are inactive at t_h.

Lemma 7.26 *Let T be a task that is not scheduled at t_h. If T is scheduled in any of the slots in $[t_b + 1, t_h)$, then T is in I.*

Proof: T clearly is not in A. Because T is scheduled in $[t_b + 1, t_h)$, T is also not in B, by Definition 7.5. ■

In the rest of this subsection, we let s denote the number of slots in $[t_b + 1, t_h)$. That is,

$$s \stackrel{\text{def}}{=} t_h - t_b - 1 \geq q + 1 \qquad \text{(by Lemma 7.25).} \qquad (7.15)$$

We now determine a lower bound on the number of subtasks of tasks in I that may be scheduled in $[t_b + 1, t_h)$ as a function of a_q^0, a_q^1, h, and s. For this purpose, we assign subtasks scheduled in $[t_b + 1, t_h)$ to processors in a systematic way. This assignment is only for accounting purposes; subtasks need not be bound to processors in the actual schedule.

Processor groups. The assignment is based on the tasks scheduled at t_h. We partition the M processors into four disjoint sets, P_1, P_2, P_3, and P_4, based on the tasks scheduled at t_h, as follows.

P_1 : By Lemma 7.21, there is at least one subtask W_ℓ scheduled at t_h such that $\mathsf{e}(W_\ell) \leq t_b$ and $\mathcal{S}(W, t) = 0$, for t in $[t_b, t_h)$. We assign one such subtask to the lone processor in this group. Hence, $|P_1| = 1$.

P_2 : The h processors that are idle at t_h comprise this group. Thus, $|P_2| = h$.

P_3 : This group consists of the $a_q^0 + a_q^1$ processors on which the a_q^0 MIs and a_q^1 SMIs are scheduled. Because either $a_q^0 > 1$ or $a_q^1 > 1$ holds, $|P_3| \geq 1$. τ^3 denotes the subset of all tasks scheduled on processors in P_3 at t_h.

P_4 : Processors not assigned to P_1, P_2, or P_3 belong to this group. τ^4 denotes the subset of all tasks scheduled on processors in P_4 at t_h.

Subtask assignment in $[t_b + 1, t_h)$. Subtasks scheduled in $[t_b + 1, t_h)$ are assigned to processors by the following rules. Tasks in τ^3 and τ^4 are assigned to the same processor that they are assigned to in t_h, in every slot in which they are scheduled in $[t_b + 1, t_h)$. (Such an assignment is possible since by the processor groups defined above, $|\tau^3| + |\tau^4| = P_3 + P_4 \leq M - h - 1 < M$.) Subtasks of tasks not in τ^3 or τ^4 may be assigned to any processor.

The next three lemmas bound the number of subtasks of tasks in I scheduled in $[t_b + 1, t_h)$. These lemmas assume that the assignment of subtasks to processors in $[t_b + 1, t_h)$ follows the rules described above. In these lemmas we assume that either $a_q^0 \geq 1$ or ($a_q^1 \geq 1$ and $a_{q-1}^0 \geq 1$) holds.

Lemma 7.27 *The number of subtasks of tasks in I that are scheduled in $[t_b + 1, t_h)$ is at least $s \cdot (h + 1) + (a_q^0 + a_q^1)$.*

Proof: We first make the following two claims.

> **Claim 7.2** *Let T_i be a subtask assigned to a processor in P_1 or P_2 in $[t_b + 1, t_h)$. Then, T is in I.*
>
> **Proof:** By our assignment of subtasks to processors, tasks assigned to processors in P_1 or P_2 in $[t_b + 1, t_h)$ are not scheduled at t_h. Therefore, T is not scheduled at t_h. Hence, by Lemma 7.26, T is inactive at t_h, *i.e.*, is in I. ■
>
> **Claim 7.3** *At least one of the subtasks assigned to each processor in P_3 in $[t_b + 1, t_h)$ is a subtask of a task in I.*

Proof: Let P_3^x be any processor in P_3, and let T_i be the subtask scheduled on P_3^x at t_h. Then, T_i is either an MI or an SMI. In the former case, by the definition of an MI, $\mathcal{S}(T, t_h - 1) = 0$, and in the latter, by the definition of an SMI, $\mathcal{S}(T, t_h - 2) = 0$. By Lemma 7.25, $t_b \leq t_h - (q + 2)$. Thus, since $q \geq 1$, and by Lemma 7.21, there is no hole in any slot in $[t_b, t_h)$, there is no hole in slot $t_h - 2$ or $t_h - 1$. Thus, a subtask of a task V other than T is assigned to P_3^x in one of these two slots. By our subtask assignment, V is not scheduled at t_h; thus, by Lemma 7.26, $V \in I$. ■

The lemma follows from the definition of s in (7.15), and Claims 7.2 and 7.3 above. ■

Lemma 7.28 *The sum of the weights of the tasks in I is at least* $\frac{(h+1)\cdot s}{s+q+1} + \frac{a_q^0 + a_q^1}{s+q+1}$.

Proof: Let V_k be a subtask of a task V in I that is scheduled in $[t_b + 1, t_h)$. Then, by Definition 3.1, $\mathsf{d}(V_k) \leq t_h$. By Definition 7.6, U_j is scheduled at t_b, and by Definition 3.2, $\mathsf{d}(U_j) \geq t_h + 1$. Because V_k with an earlier deadline than U_j is scheduled later than t_b, either $\mathsf{r}(V_k) \geq t_b + 1$ or V_k's predecessor V_j, where $j < k$, is scheduled at t_b. In the latter case, by Lemma 7.2(b), $\Delta_{\max}(V_j) \leq q$, and hence, $\mathsf{d}(V_j) \geq t_b - q + 1$, which, by Lemma 6.10, implies $\mathsf{r}(V_k) \geq t_b - q$. Thus, we have the following.

$$(\forall V_k : V \in I :: ((\exists u \mid u \in [t_b + 1, t_h) \ \wedge \ \mathcal{S}(V_k, u) = 1$$
$$\Rightarrow \ (\mathsf{r}(V_k) \geq t_b - q \ \wedge \ \mathsf{d}(V_k) \leq t_h))) \tag{7.16}$$

We next show that if $V.n$ is the number of subtasks of V scheduled in $[t_b + 1, t_h)$, then $wt(V) \geq \frac{V.n}{s+q+1}$. Let V_k and V_ℓ denote the first and final subtasks of V scheduled in $[t_b + 1, t_h)$. Then, by (7.16), $\mathsf{r}(V_k) \geq t_b - q$ and $\mathsf{d}(V_\ell) \leq t_h$. Hence,

$$\mathsf{d}(V_\ell) - \mathsf{r}(V_k) \leq t_h - t_b + q = s + q + 1 \qquad \text{(by the definition of } s \text{ in (7.15))}. \tag{7.17}$$

By (3.9) and (3.10),

$$\begin{aligned}
\mathsf{d}(V_\ell) - \mathsf{r}(V_k) &= \left\lceil \frac{\ell}{wt(V)} \right\rceil - \left\lfloor \frac{k-1}{wt(V)} \right\rfloor + \Theta(V_\ell) - \Theta(V_k) \\
&\geq \left\lceil \frac{\ell}{wt(V)} \right\rceil - \left\lfloor \frac{k-1}{wt(V)} \right\rfloor \qquad \text{(by } \ell > k \text{ and (3.8))}.
\end{aligned}$$

Hence, by (7.17), we have $\left\lceil \frac{\ell}{wt(V)} \right\rceil - \left\lfloor \frac{k-1}{wt(V)} \right\rfloor \leq s+q+1$, which implies $\frac{\ell}{wt(V)} - \frac{k-1}{wt(V)} \leq s+q+1$, *i.e.*,

$$wt(V) \geq \frac{\ell - k + 1}{s + q + 1} \geq \frac{V.n}{s + q + 1}$$

(because $V.n = \ell - k + 1$ if V is periodic and $V.n \leq \ell - k + 1$ if V is IS or GIS).

Therefore, we have $\sum_{W \in I} wt(W) \geq \sum_{W \in I} \frac{W.n}{s+q+1} \geq \frac{(h+1)\cdot s}{s+q+1} + \frac{a_1^0 + a_1^1}{s+q+1}$, where the last inequality is by Lemma 7.27. ■

Lemma 7.29 *If* $\mathsf{LAG}(\tau, t_h + 1) > \mathsf{LAG}(\tau, t_h)$ *and either* $a_q^0 \geq 1$ *or* ($a_q^1 \geq 1$ *and* $a_{q-1}^0 \geq 1$)*, then* $a_q^0 + a_q^1 \leq \min((h-1)(q+1) - 1, M - h - 1)$.

Proof: By Lemma 7.24, if $\mathsf{LAG}(\tau, t_h+1) > \mathsf{LAG}(\tau, t_h)$, then $\sum_{V \in I} wt(V) < h$. By Lemma 7.28, $\frac{(h+1)\cdot s}{s+q+1} + \frac{a_q^0 + a_q^1}{s+q+1} \leq \sum_{V \in I} wt(V)$. Therefore, $\frac{(h+1)\cdot s}{s+q+1} + \frac{a_q^0 + a_q^1}{s+q+1} < h$, which implies that

$$\begin{aligned} a_q^0 + a_q^1 &< h(q+1) - s \\ &\leq h(q+1) - (q+1) && \text{(by (7.15))} \\ &= (h-1)(q+1). && (7.18) \end{aligned}$$

Also, there are h holes in t_h, and by Lemma 7.21, $a_0 \geq 1$. Therefore, by (7.13),

$$a_q^0 + a_q^1 \leq M - h - 1. \tag{7.19}$$

(7.18) and (7.19) imply that $a_q^0 + a_q^1 \leq \min((h-1)(q+1) - 1, M - h - 1)$. ■

We now conclude Case B by establishing the following.

Lemma 7.30 *If* $a_q^0 > 0$ *or* ($a_q^1 > 0$ *and* $a_{q-1}^0 > 0$)*, then* $\mathsf{LAG}(\tau, t_h + 1) < qM + 1$.

Proof: Because $W_{\max} < 1$, assuming $W_{\max} \geq 2/3$ (because, as discussed earlier, a_0, a_{q-1}, and a_q^i are not dependent on $W_{\max}$), by (7.12), we have

$$\mathsf{LAG}(\tau, t_h + 1) < a_0 \cdot W_{\max} + ((q+1) \cdot W_{\max}) \cdot (a_q^0 + a_q^1) + (a_{q-1} + a_q^2) \cdot ((q+3) \cdot W_{\max} - 2).$$

By Lemma 7.29, if $\mathsf{LAG}(\tau, t_h + 1) > \mathsf{LAG}(\tau, t_h)$, then $a_q^0 + a_q^1 \leq \min((h-1)(q+1) - 1, M - h - 1)$. By Lemmas 7.11–7.17 (and as can be seen from the coefficients of the a_i terms in (7.20)), the lag bounds for tasks in $A_q^0 \cup A_q^1$ are higher than those for the other tasks. Hence,

$\mathsf{LAG}(\tau, t_h + 1)$ is maximized when $a_q^0 + a_q^1 = \min((h-1)(q+1)-1, M-h-1)$. We assume this is the case. Note that

$$\min((h-1)(q+1)-1, M-h-1) = \begin{cases} (h-1)(q+1)-1, & h \leq \frac{M+1+q}{q+2} \\ M-h-1, & \text{otherwise.} \end{cases} \tag{7.20}$$

Based on (7.20), we consider two cases.

Case 1: $h > \frac{M+1+q}{q+2}$. For this case, LAG is maximized when $a_q^0 + a_q^1 = M-h-1$, and hence, by (7.13), $a_0 + a_{q-1} + a_q^2 = M - h - (a_1^0 + a_1^1) = 1$. Because, by Lemma 7.21, $a_0 > 0$, we have $a_0 = 1$, and hence, $a_{q-1} = a_q^2 = 0$. Substituting $a_0 = 1$, $a_q^2 = a_{q-1} = 0$, and $a_q^1 + a_q^2 = M-h-1$ in (7.20), we have $\mathsf{LAG}(\tau, t_h+1) < W_{\max} + (q+1) \cdot W_{\max} \cdot (a_q^0 + a_q^1) = W_{\max} + (q+1) \cdot W_{\max} \cdot (M-h-1) < W_{\max} + (q+1) \cdot W_{\max} \cdot \left(M - \frac{M+q+1}{q+2} - 1\right)$ (where the last inequality is by the condition of Case 1, namely, $h > \frac{M+1+q}{q+2}$). If $qM + 1 \leq \mathsf{LAG}(\tau, t_h+1)$, then $W_{\max} + (q+1) \cdot W_{\max} \cdot \left(M - \frac{M+q+1}{q+2} - 1\right) > qM+1$, which implies that $W_{\max} > \frac{Mq(q+2)+q+2}{M(q+1)^2 - (2q^2+4q+1)}$, which is greater than $\frac{q+2}{q+3}$ for all $q \geq 1$. This contradicts (W), and hence, $\mathsf{LAG}(\tau, t_h+1) < qM+1$.

Case 2: $h \leq \frac{M+1+q}{q+2}$. For this case, LAG is maximized when $a_q^0 + a_q^1 = (h-1)(q+1)-1$. By (7.13), we have $a_{q-1} + a_q^2 = M - h - (a_0 + a_q^0 + a_q^1) = M - h - a_0 - (hq + h - q - 2)$. Therefore, by (7.20),

$$\begin{aligned} &\mathsf{LAG}(\tau, t_h+1) \\ &\quad < a_0 \cdot W_{\max} + (q+1) \cdot W_{\max} \cdot (a_q^0 + a_q^1) + ((q+3) \cdot W_{\max} - 2) \cdot (a_{q-1} + a_q^2) \\ &\quad = a_0 \cdot W_{\max} + (q+1) \cdot W_{\max} \cdot (hq + h - q - 2) \\ &\qquad +((q+3) \cdot W_{\max} - 2)(M - 2h - a_0 - hq + q + 2). \end{aligned} \tag{7.21}$$

If $qM + 1 \leq \mathsf{LAG}(\tau, t_h+1)$, then the expression on the right-hand side of (7.21) exceeds $qM+1$, which implies that $W_{\max} > \frac{(q+2)M+2q+5-4h-2a_0-2hq}{(q+3)M+2q+4-5h-(2+q)a_0-3hq}$. Let $f \stackrel{\text{def}}{=} \frac{(q+2)M+2q+5-4h-2a_0-2hq}{(q+3)M+2q+4-5h-(2+q)a_0-3hq}$, and let Y denote the denominator, $(q+3)M + 2q + 4 - 5h - (2+q)a_0 - 3hq$, of f. To show that the lemma holds for this case, we show that unless $W_{\max}$ exceeds $\frac{q+2}{q+3}$, $qM + 1 > \mathsf{LAG}(\tau, t_h+1)$. For this purpose, we determine a lower bound to the value of f. Note that for a given number of processors, M, and tardiness, q, f varies with a_0 and

h. Because $a_q^0 + a_q^1 = (h-1)(q+1)-1 > 0$, we have $h > \frac{q+2}{q+1}$; hence, because h is integral, $h \geq 2$ holds. The first derivative of f with respect to h is $\frac{M(q^2+q-2)+a_0(2q^2+2q-2)+2q^2+9q+9}{Y^2}$, which is non-negative for all $a_0 \geq 0$, and that with respect to a_0 is $\frac{M(q^2+2q-2)+h(2-2q-2q^2)+2q^2+5q+2}{Y^2}$, which is non-negative for $h \leq \frac{M(q^2+2q-2)+2q^2+5q+2}{2q^2+2q-2}$. Thus, f is minimized when $h = 2$, and because $\frac{M(q^2+2q-2)+2q^2+5q+2}{2q^2+2q-2} \geq \frac{M+1+q}{q+2}$, when $a_0 = 1$. When $h = 2$ and $a_0 = 1$ hold, $f = \frac{qM+2M-2q-5}{qM+3M-5q-8} > \frac{q+2}{q+3}$, for all M. Hence, $W_{\max} > \frac{q+2}{q+3}$, which is a violation of (W). Hence the lemma holds for this case. ■

Thus, if an MI or an SMI and a c-MI are scheduled in t_h, then (T1) is contradicted.

7.2.5.3 Case C ($A_q^0 = \emptyset$ and $A_q^1 \neq \emptyset$ and $A_{q-1}^0 = \emptyset$)

For this case, we show that if $\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) > qM + 1$, then there exists another concrete task system τ', obtained from τ by removing certain subtasks, such that LAG of τ' at $t_h - 1$ in an EPDF schedule is greater than $qM + 1$ contradicting the minimality of t_h. We begin by defining needed subsets of subtasks and tasks.

For this case, since no MI is scheduled in slot t_h, t_b (as in Definition 7.5) can be as late as $t_h - 1$. This is stated below.

$$t_b \leq t_h - 1 \tag{7.22}$$

Let t_b' be defined as follows.

Definition 7.7: t_b' denotes the latest time, if any, before $t_h - 1$ that a subtask with deadline at or after t_h is scheduled.

Since at least one SMI is scheduled at t_h, at least one MI is scheduled at $t_h - 1$. Therefore, by Lemma 7.22(a), the following holds.

> **(C)** The deadline of every subtask scheduled in any slot in $[t_h - (q+2), t_h - 1)$ is at or before $t_h - q$.

Since $q \geq 1$ holds, (C) implies

$$t_b' \leq t_h - (q+3). \tag{7.23}$$

Let τ_s^1 through τ_s^8 be subsets of subtasks defined as follows. In the definitions that follow, when we say that T_i is *ready* at t_b', we mean that $\mathsf{e}(T_i) \leq t_b'$, and T_i's predecessor, if any, is scheduled before t_b'.

$$\tau_s^1 \stackrel{\text{def}}{=} \{T_i \mid T_i \text{ is either the critical subtask at } t_h \text{ of a task in } B(t_h) \text{ or the critical subtask at } t_h - 1 \text{ of a task in } B(t_h - 1),\ t'_b \text{ exists, } T_i \text{ is scheduled at or before } t'_b, \text{ and } T \text{ is not scheduled at } t_h\}$$

$$\tau_s^2 \stackrel{\text{def}}{=} \{T_i \mid \mathsf{d}(T_i) \geq t_h,\ T_i \text{ is scheduled at } t_h - 1, \text{ and } T \text{ is not scheduled at } t_h\}$$

$$\tau_s^3 \stackrel{\text{def}}{=} \{T_i \mid T \in A_0(t_h),\ T_i \text{ is scheduled at } t_h, \text{ and } T_i \text{ is ready at or before } t_h - (q+3) \text{ in } \mathcal{S}\}$$

$$\tau_s^4 \stackrel{\text{def}}{=} \{T_i \mid T \in A_0(t_h),\ T_i \text{ is scheduled at } t_h, \text{ and } T_i \text{ is not ready at or before } t_h - (q+3) \text{ in } \mathcal{S}\}$$

$$\tau_s^5 \stackrel{\text{def}}{=} \{T_i \mid T \in (A_q^1(t_h) \cup A_q^2(t_h) \cup A_{q-1}(t_h)),\ T_i \text{ is scheduled at } t_h, \text{ and } T \text{ is scheduled at } t_h - 1\}$$

$$\tau_s^6 \stackrel{\text{def}}{=} \{T_i \mid T_i \text{ is scheduled at } t_h - 1,\ T_i \notin \tau_s^2, \text{ and } T \text{ is not scheduled at } t_h\}$$

$$\tau_s^7 \stackrel{\text{def}}{=} \{T_i \mid T_i \text{ is the predecessor of a subtask in } \tau_s^1 \text{ and } \mathsf{d}(T_h) = t_h\}$$

$$\tau_s^8 \stackrel{\text{def}}{=} \{T_i \mid T_i \text{ is the predecessor of a subtask in } \tau_s^2 \text{ and } \mathsf{d}(T_h) = t_h\}$$

Before proceeding further, we introduce some more notation. Let τ^i denote the set of all tasks with a subtask in τ_s^i, for all $1 \leq i \leq 8$. Note that $\tau^7 \subseteq \tau^1$ and $\tau^8 \subseteq \tau^2$ hold.

We establish some properties concerning the subsets of subtasks and tasks defined above.

Lemma 7.31 *The following properties hold for subsets τ_s^i and τ^i defined above, where $1 \leq i \leq 8$.*

(**a**) *For every task T, there is at most one subtask in $(\tau_s^1 \cup \tau_s^2 \cup \tau_s^6)$.*

(**b**) *Let T_i scheduled at t_h be the subtask of a task T in $A_q(t_h)$ or $A_{q-1}(t_h)$. Then, T_i is in τ_s^5.*

(**c**) *$\tau^7 \subseteq \tau^1$ and $\tau^8 \subseteq \tau^2$.*

(**d**) *Subsets τ^i, where $1 \leq i \leq 6$, are pairwise disjoint.*

Proof: Each of the above properties is proved below.

Proof of part (a). We first show that each task T has at most one subtask in τ_s^1. Let T_i in τ_s^1 be the critical subtask at t_h of T, which is in $B(t_h)$. Then, by Lemma 7.20, $\mathsf{d}(T_i) = t_h + 1$ holds. Because $wt(T) < 1$, by (3.9), $\mathsf{r}(T_i) \leq \mathsf{d}(T_i) - 2 = t_h - 1$ holds. Hence, by the definition of a critical subtask in Definition 3.2, T_i is critical at $t_h - 1$ also. Thus, if T has a critical subtask T_i at t_h and T is in $B(t_h)$, then T cannot have a subtask that is different from T_i that is critical at $t_h - 1$. Hence, it follows that each task has at most one subtask in τ_s^1.

We next show that each task can have at most one subtask in $\tau_s^2 \cup \tau_s^6$. Note that a subtask is in τ_s^2 or τ_s^6 only if it is scheduled at $t_h - 1$. Further, each task T can have at most one subtask scheduled at $t_h - 1$. Hence, if T's subtask T_i scheduled at $t_h - 1$ has its deadline at or after t_h, then T_i is in τ_s^2; else, it is in τ_s^6.

Finally, we show that if T has a subtask T_i in τ_s^1, then it does not have a subtask in $\tau_s^2 \cup \tau_s^6$, or vice versa. If T_i is in $B(t_h - 1)$, then T cannot have a subtask scheduled at $t_h - 1$, and hence, cannot have a subtask in $\tau_s^2 \cup \tau_s^6$ (because every subtask in these sets is scheduled at $t_h - 1$). On the other hand, if T_i is in $B(t_h)$ and is T's critical subtask at t_h, then note the following. **(i)** τ_s^1 is non-empty only if t'_b exists; **(ii)** by Lemma 7.20, $\mathsf{d}(T_i) = t_h + 1$ holds, and T_i is scheduled at or before t'_b, whereas a subtask in $\tau_s^2 \cup \tau_s^6$ is scheduled at $t_h - 1$. By (7.23), $t'_b \leq t_h - (q+3)$. Thus, by (ii), no subtask of T with a deadline at or before t_h can be scheduled at $t_h - 1$, and hence, can be in $\tau_s^2 \cup \tau_s^6$. On the other hand, if a subtask of T with a deadline after t_h is scheduled at $t_h - 1$, then it contradicts the fact that T_i is T's critical subtask at t_h. So, no such subtask can be in $\tau_s^2 \cup \tau_s^6$ either. ■

Proof of part (b). By the conditions of Case C, no c-MI, where $c > 0$, is scheduled at t_h. Further, because T is in $A_q(t_h)$ or $A_{q-1}(t_h)$, tardiness of T_i (scheduled at t_h) is greater than zero. Hence, by the definition of c-MI and because T is not a c-MI, T is also scheduled at $t_h - 1$. Therefore, T_i is in τ_s^5. ■

Proof of part (c). Immediate from the definitions. ■

Proof of part (d). By part (a), every task T has at most one subtask in $\tau_s^1 \cup \tau_s^2 \cup \tau_s^6$. Therefore, τ^1, τ^2, and τ_s^6 are pairwise disjoint. By the definitions of τ_s^3, τ_s^4, and τ_s^5, and by (7.2) and (7.3), A_0, A_q, and A_{q-1} are pairwise disjoint, and hence, τ_s^3, τ_s^4, and τ_s^5 are pairwise disjoint, and subtasks in them are scheduled at t_h. However, by the definitions of τ_s^1, τ_s^2, and τ_s^6, no task of a subtask in any of these subsets is scheduled at t_h. Therefore, a task in τ^1, τ^2, or τ^6 is not in $\cup_{i=3}^5 \tau^i$, that is $\tau^1 \cup \tau^2 \cup \tau^6$ is disjoint from $\cup_{i=3}^5 \tau^i$. Since τ^1, τ^2, and τ^6 are pairwise disjoint, as are τ^3, τ^4, and τ^5, all six subsets are pairwise disjoint. ■

Let τ' be a concrete GIS task system obtained from τ by removing all the subtasks in τ_s^1,

τ_s^2, τ_s^3, τ_s^7, and τ_s^8. Let $\tau_s^R = \tau_s^1 \cup \tau_s^2 \cup \tau_s^3 \cup \tau_s^7 \cup \tau_s^8$. Let $\mathcal{S}'$ be an EPDF schedule for τ' such that ties among subtasks with equal deadlines are resolved in the same way as they are resolved in $\mathcal{S}$. Our goal is to show that $\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \geq qM + 1$, and derive a contradiction to the minimality of t_h in Definition 7.3. We first establish certain properties concerning τ' and $\mathcal{S}'$.

Lemma 7.32 *No subtask with deadline at or before $t_h - 1$ is removed or displaced in $\mathcal{S}'$.*

Proof: Follows from the fact that the deadline of every subtask removed, that is, the deadline of every subtask in τ_s^R, is at or after t_h. Hence, because ties in $\mathcal{S}$ and $\mathcal{S}'$ are resolved identically, the removed subtasks cannot impact how subtasks with earlier deadlines are scheduled. (Subtasks in τ_s^1 are critical subtasks at t_h or at $t_h - 1$, and hence their deadlines are at or after t_h. Similarly, subtasks in τ_s^3 are scheduled at t_h and have a tardiness of zero, implying that their deadlines are at or after $t_h + 1$.) ■

The next few lemmas establish lag bounds for tasks with subtasks in the subsets defined above. We will denote the ideal schedule for τ as PS_τ and that for τ' as $\mathsf{PS}_{\tau'}$. We first claim the following with respect to subtasks in τ.

Claim 7.4 *The release time of every subtask in τ is at or before t_h.*

Proof: Because there is a hole in t_h (by (H)), by Lemma 6.5, no subtask scheduled at or before t_h can have a deadline after $t_h + 1$, implying that the release time of every such subtask is at or before t_h. Hence, a subtask with release time after t_h is scheduled after t_h in $\mathcal{S}$. For every such subtask, allocations in both the ideal schedule and $\mathcal{S}$ are zero in $[0, t_h + 1)$. Therefore, the LAG of τ at $t_h + 1$ does not depend on such a subtask. Further, if such a subtask is removed, the schedule before $t_h + 1$ is not impacted and no subtask scheduled at or after $t_h + 1$ can shift to t_h or earlier. Hence, the LAG of τ at $t_h + 1$ is not altered. Thus, the presence of subtasks released after t_h contradicts (T2). ■

Lemma 7.33 *Let T be a task with a subtask in τ_s^1 or τ_s^2. Then,* $\mathsf{lag}(T, t_h - 1, \mathcal{S}') = \mathsf{lag}(T, t_h + 1, \mathcal{S})$.

Proof: By (3.20),

$$\mathsf{lag}(T, t_h - 1, \mathcal{S}') = \mathsf{A}(\mathsf{PS}_{\tau'}, T, 0, t_h - 1) - \mathsf{A}(\mathcal{S}', T, 0, t_h - 1) \tag{7.24}$$

To prove this lemma, we will express the allocation to T in $\mathsf{PS}_{\tau'}$ and $\mathcal{S}'$ in terms of its allocations in PS_τ and $\mathcal{S}$, respectively. We will establish some properties needed for this purpose.

By Lemma 7.31(a), T has exactly one subtask in $\tau_s^1 \cup \tau_s^2$. Let T_i denote the distinct subtask of T that is in τ_s^1 or τ_s^2, and T_j, its predecessor in τ_s^7 or τ_s^8, respectively, if any. Note that T_j does not exist if $\mathsf{d}(T_i) = t_h$, and need not necessarily exist otherwise.

Regardless of whether T_i is in τ_s^1 or τ_s^2, T_i is scheduled at or before t'_b in $\mathcal{S}$, which by (7.23), is before $t_h - 1$. Hence, because there is a hole in t_h, by Lemma 6.5, $\mathsf{d}(T_i) \leq t_h + 1$ holds. We next show that the following holds.

(D) No subtask of T has its deadline after $t_h + 1$.

Since T is not scheduled in t_h and there is a hole in t_h, T_i's successor, if any, cannot have its eligibility time at or before t_h and deadline after $t_h + 1$. By Claim 7.4, no subtask in τ has a release time at or after $t_h + 1$. Thus, (D) holds.

We next claim that of T's subtasks, only T_i and/or T_j may receive non-zero allocations in the ideal schedule for τ in slots $t_h - 1$ and/or t_h. For this, note that the following hold: **(i)** since $\mathsf{d}(T_j) = t_h$ (by the definitions of τ_s^7 and τ_s^8), no subtask of T prior to T_j has its deadline after $t_h - 1$; **(ii)** because there is a hole in t_h, and T is not scheduled at t_h in $\mathcal{S}$ (by the definitions of τ_s^1 and τ_s^2), no subtask of T released after T has its eligibility time, and hence, release time at or before t_h. Hence, by (3.15), no subtask of T other than T_i and T_j receives any allocation in $t_h - 1$ and/or t_h. By (i) and (ii) above and because τ' contains every subtask of T that is in τ except T_i and T_j, we have $\mathsf{A}(\mathsf{PS}_{\tau'}, T, 0, t_h - 1) = \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - \mathsf{A}(\mathsf{PS}_{\tau}, T_i, 0, t_h + 1) - \mathsf{A}(\mathsf{PS}_{\tau}, T_j, 0, t_h + 1)$. Because the deadlines of T_i and T_j are at most $t_h + 1$, both these subtasks receive ideal allocations of one quantum each by $t_h + 1$. Hence,

$$\mathsf{A}(\mathsf{PS}_{\tau'}, T, 0, t_h - 1) = \begin{cases} \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - 2, & \text{if } T_j \text{ exists} \\ \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - 1, & \text{if } T_j \text{ does not exist.} \end{cases} \tag{7.25}$$

We now express the allocation to T in $\mathcal{S}'$ in terms of its allocation in $\mathcal{S}$. If T_i is in τ_s^1, then, in $\mathcal{S}$, T_i is scheduled at or before $t'_b \leq t_h - (q + 3) \leq t_h - 1$ (refer (7.23)); if it is in τ_s^2, then T_i is scheduled at $t_h - 1$. Thus, in either case, T_i is scheduled at or before $t_h - 1$ in $\mathcal{S}$. Hence, T_j, if it exists, is scheduled at or before $t_h - 1$ in $\mathcal{S}$. As for where other subtasks of T are scheduled in $\mathcal{S}$, there is a hole in t_h, and (by the definitions of τ_s^1 and τ_s^2) T is not scheduled at t_h. Therefore, if some subtask of T is scheduled after t_h, then its eligibility time is at or after $t_h + 1$, and hence its deadline is after $t_h + 1$. However, by (D), no subtask of T has a deadline after $t_h + 1$. Hence, there does not exist a subtask of T that is scheduled after t_h in

$\mathcal{S}$, which implies that there does not exist a subtask of T that is scheduled after t_h in $\mathcal{S}$ and before $t_h - 1$ in $\mathcal{S}'$. Further, because no subtask can displace to the right, there does not exist a subtask of T that is scheduled before $t_h - 1$ in $\mathcal{S}$, and at or after $t_h - 1$ in $\mathcal{S}'$. As already mentioned, every subtask of T except T_i and T_j is present in τ'. Therefore,

$$\mathsf{A}(\mathcal{S}', T, 0, t_h - 1) = \begin{cases} \mathsf{A}(\mathcal{S}, T, 0, t_h + 1) - 2, & \text{if } T_j \text{ exists} \\ \mathsf{A}(\mathcal{S}, T, 0, t_h + 1) - 1, & \text{if } T_j \text{ does not exist.} \end{cases} \tag{7.26}$$

By (7.24)–(7.26), regardless of whether T_j exists, $\mathsf{lag}(T, t_h - 1, \mathcal{S}') = \mathsf{A}(\mathsf{PS}_\tau, T, 0, t_h + 1) - \mathsf{A}(\mathcal{S}, T, 0, t_h + 1) = \mathsf{lag}(T, t_h + 1, \mathcal{S})$. ■

Lemma 7.34 *Let T be a task with a subtask in τ_s^3. Then,* $\mathsf{lag}(T, t_h - 1, \mathcal{S}') > \mathsf{lag}(T, t_h + 1, \mathcal{S}) - 1/(q+2)$.

Proof: Let T_i be T's subtask in τ_s^3. In $\mathcal{S}$, T_i is scheduled at t_h and is ready at or before $t_h - (q+3)$. Therefore, by Lemma 7.2(a), $\mathsf{r}(T_i) \le t_h - (q+3)$ holds. Since T is in $A_0(t_h)$, and T_i is scheduled at t_h in $\mathcal{S}$, the tardiness of T_i is zero in $\mathcal{S}$. Therefore, $\mathsf{d}(T_i) \ge t_h + 1$ holds, which by (H) and Lemma 7.19 implies that

$$\mathsf{d}(T_i) = t_h + 1. \tag{7.27}$$

Hence, $|\omega(T_i)| = \mathsf{d}(T_i) - \mathsf{r}(T_i) \ge q + 4$ holds, and using Lemma 3.1, it can be shown that $wt(T) < 1/(q+2)$. By Lemma 7.13, $\mathsf{lag}(T, t_h + 1, \mathcal{S}) < wt(T)$, and hence, because $wt(T) < 1/(q+2)$, it follows that

$$\mathsf{lag}(T, t_h + 1, \mathcal{S}) < 1/(q+2). \tag{7.28}$$

We next show that $\mathsf{lag}(T, t_h - 1, \mathcal{S}') = 0$. For this, we need to show that the total allocation to T in $[0, t_h - 1)$ is equal in $\mathsf{PS}_{\tau'}$ and $\mathcal{S}'$. We first show that the total allocation in $[0, t_h - 1)$ to subtasks of T released after T_i is zero in both $\mathcal{S}'$ and $\mathsf{PS}_{\tau'}$. By (7.27) and Lemma 6.10, the release time of the successor, T_j, if any, of T_i is at or after t_h. Hence, the allocation to every subtask of T released after T_i is zero in $[0, t_h - 1)$ in the ideal schedule for τ'. Also, because T_i is scheduled at t_h in $\mathcal{S}$, T_j is scheduled at or after $t_h + 1$ in $\mathcal{S}$. Hence, by Lemma 7.2(a), $\mathsf{e}(T_j) \ge t_h$ holds. Therefore, every subtask of T released after T_i is scheduled at or after t_h in $\mathcal{S}'$, that is, receives zero allocation in $[0, t_h - 1)$ in $\mathcal{S}'$.

We now show that subtasks of T released before T_i receive equal allocations in $[0, t_h - 1)$ in both $\mathsf{PS}_{\tau'}$ and $\mathcal{S}'$. Since T_i is ready at or before $t_h - (q+3)$, T_i's predecessor, if any, and all prior subtasks of T, if any, complete executing at or before $t_h - (q+3)$ in $\mathcal{S}$, and hence, in $\mathcal{S}'$, as well (because no subtask can displace to the right). Furthermore, as discussed above, $\mathsf{r}(T_i) \leq t_h - (q+3)$ holds, and hence, by Lemma 6.10, the deadline of T_i's predecessor is at or before $t_h - (q+2)$. Hence, all subtasks released before T_i complete executing by $t_h - (q+2)$ in $\mathsf{PS}_{\tau'}$ as well.

Therefore, because T_i is not present in τ', the total allocation to all the subtasks of T in τ' in $[0, t_h - 1)$ is equal in $\mathcal{S}'$ and $\mathsf{PS}_{\tau'}$. Hence, $\mathsf{lag}(T, t_h - 1, \mathcal{S}') = 0$, and because (7.28) holds, the lemma follows. ■

Lemma 7.35 *Let T be a task with a subtask in τ_s^4. Then,* $\mathsf{lag}(T, t_h - 1, \mathcal{S}') \geq \mathsf{lag}(T, t_h + 1, \mathcal{S}) - 2 \cdot W_{\max} + 1$.

Proof: First, we show that (R) holds.

(R) No subtask of T is removed.

For this, note that because T is in τ^4, by Lemma 7.31(d), T is not in τ^i, where $1 \leq i \leq 6$ and $i \neq 4$. Hence, by Lemma 7.31(c), T is also not in τ^7 or in τ^8. Thus, T does not have a subtask in τ_s^R, and hence, (R) holds.

Let T_i be T's subtask in τ_s^4 and let $t_c = t_h - (q+3)$. Then, T_i is not ready at t_c in $\mathcal{S}$. We show that T_i is not ready at t_c in $\mathcal{S}'$ also. Let T_j denote T_i's predecessor, if any, in τ.

We now show that no subtask of T that is scheduled at or after $t_h - 1$ in $\mathcal{S}$ is scheduled before $t_h - 1$ in $\mathcal{S}'$. Note that T_i is scheduled at t_h in $\mathcal{S}$. Hence, it suffices to show that T_i is not scheduled before $t_h - 1$ in $\mathcal{S}'$ (which would imply that no later subtask is scheduled before $t_h - 1$), and if T_j is scheduled at $t_h - 1$ in $\mathcal{S}$, then it is not scheduled earlier in $\mathcal{S}'$.

Because T_i is scheduled at t_h in $\mathcal{S}$ and T_i is not ready at t_c in $\mathcal{S}$, Lemma 7.2(a) implies that either $\mathsf{r}(T_i) > t_c$, or $\mathsf{r}(T_i) \leq t_c$ and T_j exists and does not complete executing by t_c. If the former holds, then because $\mathsf{r}(T_i) > t_c$ and T_i is scheduled at $t_h > t_h - (q+3) = t_c$ in $\mathcal{S}$, by Lemma 7.2(a), $\mathsf{e}(T_i) > t_c$ holds, and hence, T_i is not eligible, and hence, not ready, at t_c in $\mathcal{S}'$ either. If the latter holds, then by Lemma 6.10, $\mathsf{d}(T_j) \leq t_c + 1 \leq t_h - (q+2)$ holds, and hence, by Lemma 7.32, T_j is not displaced, and does not complete executing by t_c in $\mathcal{S}'$ also. Therefore, T_i is not ready at t_c in this case too.

Given that T_i is not ready at t_c in $\mathcal{S}'$, it is easy to show that T_i is not scheduled before $t_h - 1$ in $\mathcal{S}'$. For this, note that by Lemma 7.32, no subtask with deadline at or before $t_h - 1$ is displaced or removed. Hence, since (C) holds, no subtask scheduled in $[t_h - (q+2), t_h - 1)$

is displaced or removed. Therefore, because T_i is not ready at or before $t_c = t_h - (q+3)$, T_i cannot be scheduled before $t_h - 1$ in $\mathcal{S}'$.

We next show that if T_i's predecessor T_j exists and is scheduled at $t_h - 1$ in $\mathcal{S}$, then it is not scheduled earlier in $\mathcal{S}'$. Because T_i is scheduled at t_h and T is in $A_0(t_h)$, T_i's tardiness is zero, and hence, by Lemma 7.19, $\mathsf{d}(T_i) = t_h + 1$. Hence, $\mathsf{d}(T_j) \leq t_h$ holds. If $\mathsf{d}(T_j) < t_h$ holds, then, by Lemma 7.32, T_j is not displaced. In the other case, namely, $\mathsf{d}(T_j) = t_h$, by Lemma 6.10, $\mathsf{r}(T_i) \geq t_h - 1$, and hence, $|\omega(T_i)| = \mathsf{d}(T_i) - \mathsf{r}(T_i) \leq (t_h + 1) - (t_h - 1) = 2$ holds. Therefore, by Lemma 3.1, $|\omega(T_j)| \leq 3$, and hence, $\mathsf{r}(T_j) \geq \mathsf{d}(T_j) - 3 = t_h - 3$. If T_j is scheduled at $t_h - 1$ in $\mathcal{S}$, then by Lemma 7.2(a), $\mathsf{e}(T_j) \geq t_h - 3$. However, because $q \geq 1$, by (C), the deadline of every subtask scheduled in $[t_h - 3, t_h - 1)$ is at or before $t_h - q$, and hence, by Lemma 7.32, no such subtask is displaced or removed. Therefore, in this case too, if T_j is scheduled at $t_h - 1$ in $\mathcal{S}$, it is not scheduled earlier in $\mathcal{S}'$. Thus, no subtask of T that is scheduled at or after $t_h - 1$ in $\mathcal{S}$ is scheduled before $t_h - 1$ in $\mathcal{S}'$.

We are now ready to establish the lag of T at $t_h - 1$ in $\mathcal{S}'$. By (3.20), we have

$$
\begin{aligned}
&\mathsf{lag}(\tau', t_h - 1, \mathcal{S}') \\
&= \mathsf{A}(\mathsf{PS}_{\tau'}, T, 0, t_h - 1) - \mathsf{A}(\mathcal{S}', T, 0, t_h - 1) \\
&= \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - \mathsf{A}(\mathsf{PS}_{\tau}, T, t_h - 1, t_h + 1) \\
&\quad -(\mathsf{A}(\mathcal{S}, T, 0, t_h + 1) - \mathsf{A}(\mathcal{S}, T, t_h - 1, t_h + 1)) \\
&\qquad \text{(because, by (R), no subtask of } T \text{ is removed, and no subtask of} \\
&\qquad T \text{ scheduled at or after } t_h - 1 \text{ in } \mathcal{S} \text{ is scheduled before } t_h - 1 \text{ in } \mathcal{S}'\text{)} \\
&\geq \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - 2 \cdot W_{\max} - (\mathsf{A}(\mathcal{S}, T, 0, t_h + 1) - \mathsf{A}(\mathcal{S}, T, t_h - 1, t_h + 1)) \\
&\qquad \text{(by (3.16) and (7.10))} \\
&\geq \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - 2 \cdot W_{\max} - \mathsf{A}(\mathcal{S}, T, 0, t_h + 1) + 1 \\
&\qquad \text{(because at least one subtask of } T, T_i, \text{ is scheduled in } [t_h - 1, t_h + 1) \text{ in } \mathcal{S}\text{)} \\
&= \mathsf{lag}(T, t_h + 1, \mathcal{S}) - 2 \cdot W_{\max} + 1. \qquad \blacksquare
\end{aligned}
$$

Lemma 7.36 *Let T be a task with a subtask in τ_s^5. Then,* $\mathsf{lag}(T, t_h - 1, \mathcal{S}') \geq \mathsf{lag}(T, t_h + 1, \mathcal{S}) + 2 - 2 \cdot W_{\max}$.

Proof: As with Lemma 7.35, we first show that no subtask of T is removed. Because T is in τ^5, by Lemma 7.31(d), T is not in τ^i, where $1 \leq i \leq 6$ and $i \neq 4$. Hence, by Lemma 7.31(c), T is also not in τ^7 or in τ^8. Thus, T does not have a subtask in τ_s^R, and hence, no subtask of

T is removed.

We next show that the subtasks of T scheduled at t_h or $t_h - 1$ are not displaced.

Let T_i be T's subtask scheduled at t_h. By the definition of A_q and A_{q-1}, the tardiness of T_i is greater than zero, and hence, $\mathsf{d}(T_i) \leq t_h$. Let T_j be T_i's predecessor. By the definition of τ_s^5, T_j exists. Further, $\mathsf{d}(T_j) \leq t_h - 1$ holds and T_j is scheduled at $t_h - 1$.

We now show that T_i and T_j are not displaced. For this, observe that because $\mathsf{d}(T_j) \leq t_h - 1$ holds, T_j is not displaced by Lemma 7.32. Therefore, because T_i is T_j's successor, T_i is not ready to be scheduled until t_h, and hence, is not displaced either.

The above facts can be used to determine the lag of T at $t_h - 1$ in $\mathcal{S}'$ as follows. By (3.20), we have

$$
\begin{aligned}
&\mathsf{lag}(\tau', t_h - 1, \mathcal{S}') \\
&= \mathsf{A}(\mathsf{PS}_{\tau'}, T, 0, t_h - 1) - \mathsf{A}(\mathcal{S}', T, 0, t_h - 1) \\
&= \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - \mathsf{A}(\mathsf{PS}_{\tau}, T, t_h - 1, t_h + 1) \\
&\quad -(\mathsf{A}(\mathcal{S}, T, 0, t_h + 1) - \mathsf{A}(\mathcal{S}, T, t_h - 1, t_h + 1)) \\
&\qquad \text{(because no subtask of } T \text{ is removed, and because neither } T_i \text{ nor } T_j \text{ is} \\
&\qquad \text{displaced, no subtask of } T \text{ scheduled at or after } t_h - 1 \text{ in } \mathcal{S} \text{ is scheduled} \\
&\qquad \text{before } t_h - 1 \text{ in } \mathcal{S}') \\
&\geq \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - 2 \cdot W_{\max} \\
&\quad -(\mathsf{A}(\mathcal{S}, T, 0, t_h + 1) - \mathsf{A}(\mathcal{S}, T, t_h - 1, t_h + 1)) \quad \text{(by (3.16) and (7.10))} \\
&= \mathsf{A}(\mathsf{PS}_{\tau}, T, 0, t_h + 1) - 2 \cdot W_{\max} - \mathsf{A}(\mathcal{S}, T, 0, t_h + 1) + 2 \\
&\qquad \text{(because exactly two subtask of } T\text{, } T_i \text{ and } T_j\text{, are scheduled in } [t_h - 1, t_h + 1)) \\
&= \mathsf{lag}(T, t_h + 1, \mathcal{S}) - 2 \cdot W_{\max} + 2. \qquad \blacksquare
\end{aligned}
$$

Lemma 7.37 *Let T be a task with a subtask in τ_s^6. Then, $\mathsf{lag}(T, t_h - 1, \mathcal{S}') > \mathsf{lag}(T, t_h + 1, \mathcal{S})$.*

Proof: Let T_i denote T's subtask in τ_s^6. Because there is a hole in t_h (by (H)) and T is not scheduled at t_h, the eligibility time, and hence, the release time of T_i's successor is at least $t_h + 1$. However, by Claim 7.4, the release time of every subtask in τ is at most t_h. Therefore, T_i does not have a successor.

Since T_i is not in τ_s^2, $\mathsf{d}(T_i) \leq t_h - 1$ holds. Thus, all subtasks of T have their deadlines by $t_h - 1$ and complete executing by t_h in both PS_{τ} and $\mathcal{S}$. Therefore, T's lag at $t_h + 1$ in $\mathcal{S}$ is zero.

Because $\mathsf{d}(T_i) \leq t_h - 1$ and T_i does not have a successor, by Lemma 7.32, no subtask of T is displaced. Thus, in the ideal schedule for τ', subtasks of T complete executing by $t_h - 1$, whereas T_i is not complete until t_h in $\mathcal{S}'$. Thus, $\mathsf{lag}(T, t_h - 1, \mathcal{S}') > 0$, from which the lemma follows. ■

Let $\tau^c = \tau' \setminus (\cup_{i=1}^{6} \tau_i)$. Because τ and τ' are concrete instantiations of the same non-concrete task system, they both contain the same tasks, and hence, $\tau^c = \tau \setminus (\cup_{i=1}^{6} \tau_i)$. We show the following concerning the lag of a task in τ^c at $t_h - 1$ in $\mathcal{S}'$.

Lemma 7.38 *Let T be a task in τ^c. Then, $\mathsf{lag}(T, t_h - 1, \mathcal{S}') = \mathsf{lag}(T, t_h + 1, \mathcal{S})$.*

Proof: Because T is in τ^c, T does not contain a subtask in sets τ_s^i, where $1 \leq i \leq 8$. Hence, T does not have a subtask that is removed. We next show that T does not have a subtask that is scheduled at t_h or $t_h - 1$.

If T has a subtask T_i that is scheduled at t_h, then T is in A. By the condition of this case (Case C), $A_q^0 = \emptyset$ and $A_{q-1}^0 = \emptyset$. Hence, by (7.2), T is in one of $A_0(t_h)$, $A_q^1(t_h)$, $A_q^2(t_h)$, and $A_{q-1}^i(t_h)$, where $i \geq 1$. However, if T is in $A_0(t_h)$, then T_i is in τ_s^3 or τ_s^4. On the other hand, if T is in one of the remaining sets, then T_i has a tardiness greater than zero, but is not a c-MI, and hence, T is scheduled at $t_h - 1$; therefore, T_i is in τ_s^5. Thus, T_i is in one of τ_s^3, τ_s^4, and τ_s^5, and hence, T is in one of τ^3, τ^4, and τ^5. This contradicts the fact that T is in τ^c. Therefore, T cannot have a subtask scheduled at t_h.

We now show that T does not have a subtask scheduled at $t_h - 1$. By the definitions of τ_s^2 and τ_s^6, any subtask that is scheduled at $t_h - 1$, but does not have a later subtask of its task scheduled at t_h, is in one of these two subsets. Therefore, if T has a subtask T_i scheduled at $t_h - 1$, then because T is in τ^c (and hence not in τ^2 or τ^6), T is scheduled at t_h also. But as was shown above, T is not scheduled at t_h, and hence, is not scheduled at $t_h - 1$ either. Thus, T is not scheduled in either t_h or $t_h - 1$.

By Claim 7.4, no subtask of T is released at or after $t_h + 1$. Therefore, because there is a hole in t_h, and T is not scheduled in either t_h or $t_h - 1$, every subtask of T is scheduled before $t_h - 1$, and completes executing by $t_h - 1$ in $\mathcal{S}$. Hence, because there is a hole in t_h, by Lemma 6.5, the deadline of every subtask of T is at or before $t_h + 1$.

To complete the proof, we show that the deadline of every subtask of T is at most $t_h - 1$. Suppose to the contrary some subtask of T has its deadline after $t_h - 1$. Let T_i be such a subtask with the largest index. Then, T_i is the critical subtask of T at either t_h or $t_h - 1$ or at both times. Because T is not scheduled at either t_h or $t_h - 1$, T_i is scheduled before $t_h - 1$. Hence, T is in $B(t_h - 1)$ or $B(t_h)$ or both. Also, because $\mathsf{d}(T_i) \geq t_h$ holds, by Definition 7.7, t'_b exists and T is scheduled at or before t'_b. But then, by the definition of τ_s^1, T_i is in τ_s^1, which contradicts the fact that T_i is in τ^c. Therefore, our assumption that T has a subtask with deadline after $t_h - 1$ is incorrect.

Thus, all subtasks of T complete executing by $t_h - 1$ in both the ideal schedules. Hence, the lag of T in $\mathcal{S}$ at $t_h + 1$ is zero.

Because no subtask of T is removed or displaced, and every subtask of T is scheduled before $t_h - 1$ in $\mathcal{S}$, all subtasks of T complete executing by $t_h - 1$ in $\mathcal{S}'$ also. Therefore, $\mathsf{lag}(T, t_h - 1, \mathcal{S}') = 0$. The lemma follows. ■

Having determined bounds for the lags of tasks at $t_h - 1$ in $\mathcal{S}'$, we now determine a lower bound for the LAG of τ' at $t_h - 1$ in $\mathcal{S}'$, and show that $\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \geq qM + 1$.

Lemma 7.39 *If either $W_{\max} \leq \frac{q+3}{2q+4}$ and $a_0 \leq \frac{(M-h)\cdot(q+1)}{q+2}$ or $W_{\max} > \frac{q+3}{2q+4}$ and $a_0 \leq 2(M - h)(1 - W_{\max})$, then $\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \geq qM + 1$.*

Proof: By (3.26),

$$\begin{aligned}
&\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \\
&= \sum_{T \in \tau'} \mathsf{lag}(T, t_h - 1, \mathcal{S}') \\
&= \sum_{T \in \tau} \mathsf{lag}(T, t_h - 1, \mathcal{S}') \qquad \text{(by the construction of } \tau'\text{)} \\
&= \sum_{i=1}^{6} \sum_{T \in \tau^i} \mathsf{lag}(T, t_h - 1, \mathcal{S}') + \sum_{T \in \tau^c} \mathsf{lag}(T, t_h - 1, \mathcal{S}') \\
&\qquad \text{(by Lemmas 7.31(c) and (d), and because } \tau^c = \tau \setminus \cup_{i=1}^{6} \tau^i\text{)} \\
&\geq \sum_{T \in \tau^1 \cup \tau^2 \cup \tau^6 \cup \tau^c} \mathsf{lag}(T, t_h + 1, \mathcal{S}) + \sum_{i=3}^{5} \sum_{T \in \tau^i} \mathsf{lag}(T, t_h - 1, \mathcal{S}') \\
&\qquad \text{(by Lemmas 7.33, 7.37, and 7.38)} \\
&\geq \sum_{i=1}^{6} \sum_{T \in \tau^i} \mathsf{lag}(T, t_h + 1, \mathcal{S}) + \sum_{T \in \tau^c} \mathsf{lag}(T, t_h + 1, \mathcal{S}) - |\tau^3| \cdot \frac{1}{q+2} \\
&\quad + |\tau^4| \cdot (1 - 2W_{\max}) + |\tau^5| \cdot (2 - 2W_{\max}) \qquad \text{(by Lemmas 7.34–7.36)} \\
&= \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - |\tau^3| \cdot \frac{1}{q+2} - |\tau^4| \cdot (2W_{\max} - 1) + |\tau^5| \cdot (2 - 2W_{\max}) \qquad (7.29) \\
&\qquad \text{(by the definitions of sets } \tau^i\text{, where } 1 \leq i \leq 6\text{, and } \tau^c\text{).}
\end{aligned}$$

Note that

$$|\tau^3| + |\tau^4| = a_0. \tag{7.30}$$

By Lemma 7.31(b), $|\tau^5| = |A_q| + |A_{q-1}| = a_q + a_{q-1}$. By the definitions of A_q, A_q^0, A_q^1, and A_q^2, and by (7.2)–(7.4), $a_q = a_q^0 + a_q^1 + a_q^2$. However, because no MI is scheduled at t_h by the conditions of Case C, $a_q^0 = 0$, and hence,

$$|\tau^5| = a_q^1 + a_q^2 + a_{q-1} = M - h - a_0 \qquad \text{(by (7.13))}. \tag{7.31}$$

We now consider the following two cases based on the statement of the lemma.

Case 1: $\boldsymbol{W_{\max} > \frac{q+3}{2q+4}}$ and $\boldsymbol{a_0 \leq 2(M-h)(1-W_{\max})}$. Since $W_{\max} > \frac{q+3}{2q+4}$, $2W_{\max} - 1 > \frac{1}{q+2}$ holds. By (7.29),

$$\begin{aligned}
&\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \\
&\quad \geq \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - |\tau^3| \cdot \frac{1}{q+2} - |\tau^4| \cdot (2W_{\max} - 1) + |\tau^5| \cdot (2 - 2W_{\max}) \\
&\quad \geq \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - |\tau^3| \cdot (2W_{\max} - 1) - |\tau^4| \cdot (2W_{\max} - 1) + |\tau^5| \cdot (2 - 2W_{\max}) \\
&\qquad\qquad \text{(because as mentioned above, } 2W_{\max} - 1 > \tfrac{1}{q+2}\text{)} \\
&\quad = \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - a_0 \cdot (2W_{\max} - 1) + (M - h - a_0) \cdot (2 - 2W_{\max}) \\
&\qquad\qquad \text{(by (7.30) and (7.31))} \\
&\quad = \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - a_0 + (M - h) \cdot (2 - 2W_{\max}) \\
&\quad \geq \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) \qquad \text{(because } 2(M-h) \cdot (1 - W_{\max}) \geq a_0 \text{ for this case)} \\
&\quad \geq qM + 1 \qquad \text{(by (T1))}.
\end{aligned} \tag{7.32}$$

Case 2: $\boldsymbol{W_{\max} \leq \frac{q+3}{2q+4}}$ and $\boldsymbol{a_0 \leq \frac{(M-h)\cdot(q+1)}{q+2}}$. Since $W_{\max} \leq \frac{q+3}{2q+4}$, $2 \cdot W_{\max} - 1 \leq \frac{1}{q+2}$ holds. As with Case 1, by (7.29),

$$\begin{aligned}
&\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \\
&\quad \geq \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - |\tau^3| \cdot \frac{1}{q+2} - |\tau^4| \cdot (2W_{\max} - 1) + |\tau^5| \cdot (2 - 2 \cdot W_{\max}) \\
&\quad \geq \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - |\tau^3| \cdot \frac{1}{q+2} - |\tau^4| \cdot \frac{1}{q+2} + |\tau^5| \cdot (2 - 2 \cdot W_{\max}) \\
&\qquad\qquad \text{(because } 2 \cdot W_{\max} - 1 \leq \tfrac{1}{q+2}\text{)} \\
&\quad \geq \mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) - |\tau^3| \cdot \frac{1}{q+2} - |\tau^4| \cdot \frac{1}{q+2} + |\tau^5| \cdot \frac{2q+2}{2(q+2)}
\end{aligned}$$

$$
\begin{aligned}
& \qquad\qquad \text{(because } W_{\max} \leq \tfrac{q+3}{2(q+2)}\text{)} \\
= \;& \mathsf{LAG}(\tau, t_h+1, \mathcal{S}) - a_0 \cdot \frac{1}{q+2} + (M-h-a_0) \cdot \frac{q+1}{q+2} \\
& \qquad\qquad \text{(by (7.30) and (7.31))} \\
= \;& \mathsf{LAG}(\tau, t_h+1, \mathcal{S}) - a_0 + (M-h) \cdot \frac{q+1}{q+2} \\
\geq \;& \mathsf{LAG}(\tau, t_h+1, \mathcal{S}) \qquad \text{(because } a_0 \leq \tfrac{(M-h)\cdot(q+1)}{q+2} \text{ for this case)} \\
\geq \;& qM+1 \qquad \text{(by (T1)).} \qquad (7.33)
\end{aligned}
$$

The lemma follows from (7.32) and (7.33), and by the conditions of Cases 1 and 2, respectively. ■

In completing Case C, we make use of this auxiliary algebraic lemma.

Lemma 7.40 *The roots of* $f(W_{\max}) \stackrel{\text{def}}{=} 2(M-h)(q+1)W_{\max}^2 - (q+2)(M-h)W_{\max} - ((q-1)M+1+h) = 0$ *are* $W_{\max} = \frac{(q+2)(M-h)\pm\sqrt{9q^2(M-h)^2+\Delta}}{4(M-h)(q+1)}$*, where* $\Delta = 4(M-h)(M(q-1)+h(2q^2+q+1)+2q+2)$.

Proof: The roots of $f(W_{\max})$ are given by $\frac{(q+2)(M-h)\pm\sqrt{(q+2)^2(M-h)^2+8(M-h)(q+1)((q-1)M+1+h)}}{4(M-h)(q+1)}$. Let $I = (q+2)^2(M-h)^2 + 8(M-h)(q+1)((q-1)M+1+h)$ (the term within the square root). Then,

$$
\begin{aligned}
I \;=\; & (q+2)^2(M-h)^2 + 8(M-h)(q+1)((q-1)M+1+h) \\
=\; & q^2(M-h)^2 + (4q+4)(M-h)^2 + 8q^2(M-h)^2 - 8q^2(M-h)^2 \\
& +8(M-h)(q+1)((q-1)M+1+h) \\
& \qquad \text{(splitting the first term, and adding and subtracting } 8q^2(M-h)^2\text{)} \\
=\; & 9q^2(M-h)^2 + 4(M-h)(M(q-1)+h(2q^2+q+1)+2q+2) \\
=\; & 9q^2(M-h)^2 + \Delta. \qquad ■
\end{aligned}
$$

We conclude this case by establishing the following lemma.

Lemma 7.41 *If either* $W_{\max} \leq \frac{q+3}{2q+4}$ *and* $a_0 > \frac{(M-h)\cdot(q+1)}{q+2}$ *or* $W_{\max} > \frac{q+3}{2q+4}$ *and* $a_0 > 2(M-h)(1-W_{\max})$*, then* $\mathsf{LAG}(\tau, t_h+1, \mathcal{S}) < qM+1$.

Proof: We consider two cases based on the statement of the lemma.

Case 1: $\boldsymbol{W_{\max} > \frac{q+3}{2q+4}}$ and $\boldsymbol{a_0 > 2(M-h)(1-W_{\max})}$. By (7.11),

$$
\begin{aligned}
&\mathsf{LAG}(\tau, t_h+1, \mathcal{S}) \\
&\quad < \; a_0 \cdot W_{\max} + a_q^0(q+1)W_{\max} + a_{q-1} \cdot q \cdot W_{\max} + a_q^1((q+2)W_{\max} - 1) \\
&\qquad +a_q^2((q+3)W_{\max} - 2) \\
&\quad < \; a_0 \cdot W_{\max} + a_q^0(q+1)W_{\max} + (a_{q-1} + a_q^1)((q+2)W_{\max} - 1) + a_q^2((q+3)W_{\max} - 2) \\
&\qquad\quad \text{(by the conditions of Case 1, } W_{\max} > \tfrac{q+3}{2q+4} \geq \tfrac{1}{2}\text{; thus, } qW_{\max} < (q+2)W_{\max} - 1\text{)} \\
&\quad < \; a_0 \cdot W_{\max} + a_q^0(q+1)W_{\max} + (a_{q-1} + a_q^1 + a_q^2)((q+2)W_{\max} - 1) \qquad \text{(as } W_{\max} < 1\text{)} \\
&\quad \leq \; a_0 \cdot W_{\max} + (M - h - a_0) \cdot ((q+2)W_{\max} - 1) \\
&\qquad\qquad \text{(by (7.13) because } a_q^0 = 0 \text{ by the conditions of Case C)} \\
&\quad = \; a_0 \cdot (1 - (q+1)W_{\max}) + (M-h) \cdot ((q+2)W_{\max} - 1) \\
&\quad < \; 2(M-h)(1-W_{\max}) \cdot (1 - (q+1)W_{\max}) + (M-h) \cdot ((q+2)W_{\max} - 1) \\
&\qquad\qquad \text{(because } W_{\max} > \tfrac{q+3}{2q+4} \geq \tfrac{1}{q+1} \text{ for all } q \geq 1\text{, } 1 - (q+1)W_{\max} < 0\text{; also, by the} \\
&\qquad\qquad \text{conditions of Case 1, } a_0 > 2(M-h)(1-W_{\max})\text{)} \\
&\quad = \; 2(M-h)(q+1)W_{\max}^2 - (q+2)(M-h)W_{\max} + M - h.
\end{aligned}
$$

We next show that $\mathsf{LAG}(\tau, t_h+1, \mathcal{S}) < qM + 1$ holds (if (W) holds). Suppose to the contrary that $\mathsf{LAG}(\tau, t_h+1, \mathcal{S}) \geq qM + 1$; then by the derivation above

$$2(M-h)(q+1)W_{\max}^2 - (q+2)(M-h)W_{\max} - ((q-1)M + 1 + h) > 0. \tag{7.34}$$

By Lemma 7.40, the roots of $f(W_{\max}) = 2(M-h)(q+1)W_{\max}^2 - (q+2)(M-h)W_{\max} - ((q-1)M + 1 + h) = 0$ are $W_{\max} = \frac{(q+2)(M-h) \pm \sqrt{9q^2(M-h)^2 + \Delta}}{4(M-h)(q+1)}$, where $\Delta = 4(M-h)(M(q-1) + h(2q^2+q+1) + 2q + 2)$. Let $W_{\max,1} = \frac{(q+2)(M-h) + \sqrt{9q^2(M-h)^2 + \Delta}}{4(M-h)(q+1)}$ and $W_{\max,2} = \frac{(q+2)(M-h) - \sqrt{9q^2(M-h)^2 + \Delta}}{4(M-h)(q+1)}$. Since $0 < h < M$ and $q \geq 1$ hold, $\Delta > 0$ holds, and hence, $\sqrt{9q^2(M-h)^2 + \Delta}$ is greater than $3q(M-h)$. Note that $W_{\max,1} > \frac{(q+2)(M-h) + 3(M-h)q}{4(M-h)(q+1)} = \frac{4q+2}{4q+4} > 0$. Also, $3q(M-h) \geq (q+2)(M-h)$ for all $q \geq 1$. Therefore, $W_{\max,2} < 0$. The first derivative of $f(W_{\max})$ with respect to $W_{\max}$ is given by $f'(W_{\max}) = 4(M-h)(q+1)W_{\max} - (q+2)(M-h)$, which is positive for $W_{\max} > \frac{q+2}{4q+4}$. Hence, $f(W_{\max})$ is an increasing function of $W_{\max}$ for $W_{\max} \geq \frac{q+2}{4q+4}$; further, the following hold: $W_{\max,1} > \frac{4q+2}{4q+4} > \frac{q+2}{4q+4}$, $f(W_{\max,1}) = 0$, and $f(W_{\max})$ is quadratic. Therefore, we have $f(W_{\max}) < 0$ for $W_{\max,2} < 0 < W_{\max} < W_{\max,1}$. Because as mentioned earlier, $W_{\max,1} > \frac{(q+2)(M-h) + 3(M-h)q}{4(M-h)(q+1)} = \frac{4q+2}{4q+4} > \frac{q+2}{q+3}$, it follows that, for all $0 < W_{\max} \leq \frac{q+2}{q+3}$,

$f(W_{\max}) < 0$. By (W), $W_{\max} \leq \frac{q+2}{q+3}$ holds, and hence, (7.34) does not hold, implying that $\mathsf{LAG}(\tau, t_h + 1) < qM + 1$. Thus, by the conditions of Case 1, if $W_{\max} > \frac{q+3}{2q+4}$ and $a_0 > 2(M - h)(1 - W_{\max})$, then $\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) < qM + 1$ follows.

Case 2: $\boldsymbol{W_{\max} \leq \frac{q+3}{2q+4}}$ and $\boldsymbol{a_0 > \frac{(M-h)\cdot(q+1)}{q+2}}$. Because $\frac{q+3}{2q+4} \leq \frac{2}{3}$, for all $q \geq 1$, $q \cdot W_{\max} \geq (q+3)W_{\max} - 2$ holds. Hence, by (7.11), we have

$$\begin{aligned}
&\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) \\
&\quad < \quad a_0 \cdot W_{\max} + (a_{q-1} + a_q^2)q \cdot W_{\max} + a_q^0(q+1)W_{\max} + a_q^1((q+2)W_{\max} - 1) \\
&\quad = \quad a_0 \cdot W_{\max} + (a_{q-1} + a_q^2)q \cdot W_{\max} + a_q^1((q+2)W_{\max} - 1) \qquad (7.35) \\
&\qquad\qquad \text{(because } a_q^0 = 0 \text{ by the conditions of Case C).}
\end{aligned}$$

We consider two subcases based on the value of $W_{\max}$.

Subcase 2(a): $\boldsymbol{\frac{1}{2} < W_{\max} \leq \frac{q+3}{2q+4}}$. For this case, $(q+2)W_{\max} - 1 > q \cdot W_{\max}$ holds. Hence, by (7.35), we have

$$\begin{aligned}
&\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) \\
&\quad < \quad a_0 \cdot W_{\max} + (a_{q-1} + a_q^2 + a_q^1)((q+2)W_{\max} - 1) \\
&\quad \leq \quad a_0 \cdot W_{\max} + (M - h - a_0) \cdot ((q+2)W_{\max} - 1) \qquad \text{(by (7.13) because } a_q^0 = 0) \\
&\quad = \quad a_0 \cdot (1 - (q+1)W_{\max}) + (M - h) \cdot ((q+2)W_{\max} - 1). \qquad (7.36)
\end{aligned}$$

Let $f(a_0, W_{\max}) \stackrel{\text{def}}{=} a_0 \cdot (1 - (q+1)W_{\max}) + (M-h) \cdot ((q+2)W_{\max} - 1)$, the right-hand side of the above inequality. Our goal is to determine an upper bound for $f(a_0, W_{\max})$. We first show that $f(a_0, W_{\max})$ is an increasing function of $W_{\max}$ for all $a_0 \geq 0$, and a decreasing function of a_0, for any $W_{\max} \geq \frac{1}{q+1}$. (In the description that follows, we assume a_0 and $W_{\max}$ are non-negative.) The first derivative of $f(a_0, W_{\max})$ with respect to $W_{\max}$ is $(M-h)(q+2) - a_0(q+1)$. Therefore, since $a_0 \leq M - h$ and $M - h > 0$, it follows that $(M-h)(q+2) - a_0(q+1)$ is positive for all $q \geq 0$. Hence, $f(a_0, W_{\max})$ is an increasing function of $W_{\max}$ for all a_0. Further, $f(a_0, W_{\max})$ is a non-decreasing function of a_0 for all $W_{\max} \leq \frac{1}{q+1}$, and is a decreasing function of a_0 for all $W_{\max} > \frac{1}{q+1}$. Therefore, since $W_{\max} \leq \frac{q+3}{2q+4}$, $a_0 \leq M - h$, and $a_0 \geq 1$ (by Lemma 7.21), $f(a_0, W_{\max})$ is maximized when either $W_{\max} = \frac{q+3}{2q+4}$ and $a_0 = 1$ or $W_{\max} = \frac{1}{q+1}$ and $a_0 = M - h$. It can easily be verified that $f(a_0, \frac{q+3}{2q+4}) = a_0 \cdot \left(\frac{-q^2-2q+1}{2q+4}\right) + M \cdot \left(\frac{q+1}{2}\right) - h \cdot \left(\frac{q+1}{2}\right) < qM + 1$

for all $a_0 \geq 1$. It can also be verified that $f(a_0, \frac{1}{q+1}) = \frac{M-h}{q+1} < qM + 1$ for all a_0. Hence, $f(a_0, W_{\max}) < qM + 1$, and therefore, $\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) < qM + 1$ holds.

Subcase 2(b): $\boldsymbol{W_{\max} \leq \frac{1}{2}}$. For this case, $(q+2)W_{\max} - 1 \leq q \cdot W_{\max}$ holds. Hence, by (7.35), we have

$$\begin{aligned}
&\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) \\
&\quad < \; a_0 \cdot W_{\max} + (a_{q-1} + a_q^1 + a_q^2) \cdot q \cdot W_{\max} \\
&\quad \leq \; a_0 \cdot W_{\max} + (M - h - a_0) \cdot q \cdot W_{\max} && \text{(by (7.13) because } a_q^0 = 0) \\
&\quad = \; a_0 \cdot W_{\max}(1 - q) + (M - h) \cdot q \cdot W_{\max} \\
&\quad \leq \; (M - h) \cdot q \cdot W_{\max} && \text{(because } q \geq 1) \\
&\quad < \; qM + 1.
\end{aligned}$$

By the reasoning in subcases 2(a) and 2(b), it follows that if $W_{\max} \leq \frac{q+3}{2q+4}$ and $a_0 \geq \frac{(M-h)\cdot(q+1)}{q+2}$, then $\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) < qM + 1$.

Finally, the lemma holds by the conclusions drawn in Cases 1 and 2. ■

By Lemmas 7.39 and 7.41, for any a_0 and $W_{\max}$, either $\mathsf{LAG}(\tau, t_h + 1, \mathcal{S}) < qM + 1$ or $\mathsf{LAG}(\tau', t_h - 1, \mathcal{S}') \geq qM + 1$ holds. Thus, either (T1) or Definition 7.3 is contradicted.

7.2.5.4 Case D ($A_q^0 = A_q^1 = \emptyset$)

Lemma 7.42 *If $A_q^0 = A_q^1 = \emptyset$, then $\mathsf{LAG}(\tau, t_h + 1) < qM + 1$.*

Proof: Because $a_q^0 = a_q^1 = 0$, and a_0, a_{q-1}, and a_q^2 are independent of $W_{\max}$, as explained earlier (when (7.12) was established), we bound $\mathsf{LAG}(\tau, t_h + 1)$ assuming $W_{\max} \geq 2/3$. Hence, by (7.12), and $A_q^0 = A_q^1 = \emptyset$, we have $\mathsf{LAG}(\tau, t_h + 1) < a_0 \cdot W_{\max} + ((q+3)W_{\max} - 2) \cdot (a_q^2 + a_{q-1})$, which, by (7.13), equals $a_0 \cdot W_{\max} + ((q+3)W_{\max} - 2) \cdot (M - h - a_0)$.

Contrary to the statement of the lemma, assume $\mathsf{LAG}(\tau, t_h + 1) \geq qM + 1$. This assumption implies that $a_0 \cdot W_{\max} + ((q+3)W_{\max} - 2) \cdot (M - h - a_0) > qM + 1$, which, in turn, implies that $W_{\max} > f \stackrel{\text{def}}{=} \frac{(q+2)M - 2h - 2a_0 + 1}{(q+3)M - (q+3)h - (q+2)a_0}$. We now determine a lower bound for f and show that f lies outside the range of values assumed for $W_{\max}$ and arrive at a contradiction. Let Y denote the denominator of f. The first derivative of f with respect to h is given by $\frac{q(q+3)M - 2a_0 + q + 3}{Y^2}$, which is non-negative for all $M \geq 1$, $a_0 \geq 1$, and $q \geq 1$. The first derivative of f with respect to a_0 is given by $\frac{M(q^2+q-2) + 2h + q + 2}{Y^2}$, which is also non-negative for all $M \geq 1$, $q \geq 1$, and

$h \geq 0$. Hence, since h and a_0 are greater than zero, f is minimized when $h = a_0 = 1$, for which $f = \frac{(q+2)M-3}{(q+3)M-2q-5} > \frac{q+2}{q+3}$ holds, for all $q \geq 1$, $M > 1$. This violates (W), and hence, our assumption is false, and the lemma follows. ■

By Lemmas 7.23, 7.30, 7.39, 7.41, and 7.42, if (W) is satisfied, then either $\mathsf{LAG}(\tau, t_h+1) < qM+1$ or there exists another task system with LAG under EPDF at least $qM+1$ at $t_h - 1$. Thus, either the minimality of t_h or (T1) is contradicted. So, task system τ as defined in Definition 7.4 does not exist, and Theorem 7.2 holds.

The following corollary follows easily from Theorem 7.2.

Corollary 7.1 *If the weight of each task in a feasible GIS task system τ is at most $W_{\max}$, then* EPDF *ensures a tardiness bound of* $\max(1, \left\lceil \frac{3 \cdot W_{\max}-2}{1-W_{\max}} \right\rceil)$ *for τ.*

Proof: Assume to the contrary that the tardiness for some subtask in τ is q, where $q > \max(1, \left\lceil \frac{3 \cdot W_{\max}-2}{1-W_{\max}} \right\rceil)$. Then, $q > \max(1, \frac{3 \cdot W_{\max}-2}{1-W_{\max}})$ holds, which implies that $q > 1$ and $W_{\max} < \frac{q+2}{q+3}$. This contradicts Theorem 7.2. ■

7.3 A Sufficient Restriction on Total System Utilization for Bounded Tardiness

In the previous section, we determined a sufficient restriction on per-task weights for guaranteeing bounded tardiness under EPDF, assuming that the total system utilization, U_{sum}, is not restricted (except not exceeding the number of processors, M) (Result 1). In Chapter 6, we determined a sufficient restriction on total system utilization for ensuring that no deadline is missed under EPDF, given the maximum weight, $W_{\max}$, of any task, including $W_{\max} = 1$ (Result 2). In this section, we provide a simple extension to the analysis developed in Chapter 6 to determine a sufficient restriction on U_{sum} for ensuring a tardiness bound of q quanta when per-task weights are not capped (Result 3).

One may reasonably question whether a separate analysis is needed for each of the three results and whether a generalized analysis from which each of the three results can be deduced as special cases is not possible. Our answer to this question is as follows. Result 1, presented in this chapter, required analyzing four different cases in order to arrive at per-task utilization restrictions that are reasonably liberal. However, the analysis was centered around a single slot with a hole across which LAG increases and exploited the fact that the maximum task weight is capped. On the other hand, the analysis for Results 2 and 3 hinges on determining the number of slots with no holes that precede or follow a slot with a hole across which LAG increases, and bounding the decrease in LAG that is possible in those slots. (because the total system utilization may be less than the number of processors). Hence, establishing a reasonable upper bound on LAG requires considering more than one slot with a hole across which LAG increases. Detailed case analyses, as was considered in the previous section, becomes intractable when it has to be combined with reasoning that is possible when total utilization is capped. Similarly, Results 2 and 3 each consists of analysis that is specific to it, and hence, it is much cleaner to keep them both separate.

As mentioned above, in this section, we establish a sufficient restriction on total system utilization, which is given by Theorem 7.3, for ensuring a tardiness bound of q quanta under EPDF. For this section, let (W) be as defined below. (To simplify the analysis, we restrict the maximum weight of a task to be strictly less than one. The result can be shown to hold even when $W_{\max} = 1$ holds at the expense of some extra analysis that was used for the problem considered in the previous chapter.)

(W) The weight of each task in the task system under consideration is less than $W_{\max} < 1$, and the sum of the weights of all the tasks is at most $\min(M, \frac{((q+1)W_{\max}+(q+2))M+((2q+1)W_{\max}+1)}{2(q+1)W_{\max}+2})$.

Theorem 7.3 *Tardiness under* EPDF *is at most q quanta, where $q \geq 1$, for every GIS task system satisfying* (W).

Our setup is similar to that used in Section 7.2. Assume that Theorem 7.3 does not hold. Hence, there exists a time t_d as defined in Definition 7.1.

Definition 7.8: τ is a concrete GIS task system satisfying (W), (S1), (S2), and (S3), where (S1) and (S2) are as defined in Section 7.2 (but with respect to τ), (W) is as defined in this section, and (S3) is as defined below.

(S3) No task system satisfying (W), (S1), and (S2) has a larger rank than τ.

Recall that the rank of a task system refers to the sum of the eligibility times of all its subtasks, *i.e.*, $rank(\tau, t) = \sum_{\{T_i \in \tau\}} \mathsf{e}(T_i)$.

As mentioned in Section 7.2, though there are some differences in the task system considered here from those considered in the previous section and in the context of other problems, some basic properties can be shown to hold for all these task systems, each of which is defined as a minimal system violating a theorem to be proved. We borrow such properties without proof.

In what follows, let $\mathcal{S}$ denote an EPDF schedule for τ in which a subtask of τ with deadline at t_d has a tardiness of $q+1$. The following claim and lemma are counterparts of Claim 7.1 and Lemma 7.1. Their proofs are also identical with the exception of substituting τ for σ and $\mathcal{S}$ for $\mathcal{S}'$.

Claim 7.5 *There is no hole in any slot in* $[t_d - 1, t_d + q)$ *in* $\mathcal{S}$.

Lemma 7.43 $\mathsf{LAG}(\tau, t_d, \mathcal{S}) = qM + 1$.

Based on the above, we establish Claim 7.6 and Lemma 7.44. As in Chapter 6, let α be defined as follows. (α denotes the total utilization of τ, expressed as a fraction of M.)

$$\alpha \stackrel{\text{def}}{=} \frac{\sum_{T \in \tau} wt(T)}{M} \tag{7.37}$$

Claim 7.6 *There is no hole in slot* $t_d - 2$.

Proof: By Claim 7.5, there is no hole in slot $t_d - 1$. The deadline of every subtask that is scheduled at $t_d - 1$ is at or before t_d (otherwise, such a subtask can be removed, contradicting (S2)). By (W), the weight of every task in τ is less than one. Hence, by Lemma 3.1, $|\omega(T_i)| \geq 2$ holds for every subtask T_i in τ. Therefore, the release time of each of the M subtasks scheduled in slot $t_d - 1$ is at or before $t_d - 2$, and hence, each such subtask is eligible at $t_d - 2$. Thus, if there is a hole in $t_d - 2$, then it contradicts the fact that EPDF is work conserving. ■

Lemma 7.44 $\mathsf{LAG}(\tau, t_d - 2, \mathcal{S}) = M(q+2) - 2\alpha \cdot M + 1$.

Proof: By (3.31) (with t' and $t+1$ as $t_d - 2$ and t_d, respectively),

$$\begin{aligned}
&\mathsf{LAG}(\tau, t_d - 2) \\
&\quad \geq \ \mathsf{LAG}(\tau, t_d) - (t_d - (t_d - 2)) \cdot \textstyle\sum_{T \in \tau} wt(T) + \sum_{u=t_d-2}^{t_d-1} \sum_{T \in \tau} \mathcal{S}(T, u) \\
&\quad = \ \mathsf{LAG}(\tau, t_d) - 2\alpha M + 2M \\
&\qquad\quad \text{(by (7.37) and as there is no hole in slots } t_d - 2 \text{ and } t_d - 1 \text{ by Claims 7.5 \& 7.6)} \\
&\quad = \ qM + 1 + 2M - 2\alpha \cdot M \qquad \text{(by Lemma 7.43).}
\end{aligned}$$

■

By Lemma 7.44 above, $\mathsf{LAG}(\tau, t_d-2, \mathcal{S}) = (q+2)M - 2\alpha \cdot M + 1$. Since LAG is zero at time zero, this implies that there exists some time t_L such that t_L is the earliest time slot across which LAG increases to at least $(q+2)M - 2\alpha \cdot M + 1$. This is stated below.

$$\begin{aligned}
0 \le t_L < t_d - 2 \quad &\wedge \\
(\forall t : 0 \le t \le t_L :: \mathsf{LAG}(\tau, t, \mathcal{S}) < (q+2)M - 2\alpha \cdot M + 1) \quad &\wedge \\
\mathsf{LAG}(\tau, t_L + 1, \mathcal{S}) \ge (q+2)M - 2\alpha \cdot M + 1 \quad & \qquad (7.38)
\end{aligned}$$

Our goal is to contradict the existence of t_L, specifically, to show that there does not exist a time at which LAG in $\mathcal{S}$ exceeds or equals $(q+2)M - 2\alpha M + 1$, and thereby, derive a contradiction to Lemma 7.44, and hence, to the existence of τ. Theorem 7.3 will then follow.

By Lemma 6.12(b) proved in Chapter 6, if $\mathsf{LAG}(\tau, t+1) \ge \mathsf{LAG}(\tau, t-\lambda+2)$, where $\lambda = \max(2, \left\lceil \frac{1}{W_{\max}} \right\rceil)$, then there is no hole in slot $t-\lambda+1$. If $W_{\max} \le \frac{q+2}{q+3}$, then by Theorem 7.2, tardiness is at most one quantum even if the total utilization is M. Hence, assume $W_{\max} > \frac{q+2}{q+3} > \frac{1}{2}$, which implies that $\lambda = 2$. By Lemma 6.11 of Chapter 6, $\mathsf{LAG}(\tau, 1) \le \mathsf{LAG}(\tau, 0)$. Thus, we have the following corollary.

Corollary 7.2 *If* $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t)$*, then* $t \ge 1$*, and there is no hole in slot* $t-1$*.*

Lemma 7.45 *Let* $t \le t_L$ *be a slot across which* LAG *increases. Then,* $\mathsf{LAG}(\tau, t+1) < (2+q)M - 2\alpha M + 1$*.*

Proof: Let x denote the number of tasks scheduled in t. Because there is an increase in LAG across t, by Corollary 7.2, slot $t-1$ exists and there is no hole there. Therefore, by by (3.31),

$$\begin{aligned}
\mathsf{LAG}(\tau, t+1) &\le \mathsf{LAG}(\tau, t-1) + (t+1-(t-1)) \cdot \sum_{T \in \tau} wt(T) \\
&\quad - \sum_{T \in \tau} \mathcal{S}(T, t-1) - \sum_{T \in \tau} \mathcal{S}(T, t) \\
&= \mathsf{LAG}(\tau, t-1) + 2 \cdot \sum_{T \in \tau} wt(T) - \sum_{T \in \tau} \mathcal{S}(T, t-1) - \sum_{T \in \tau} \mathcal{S}(T, t) \\
&\le \mathsf{LAG}(\tau, t-1) + 2\alpha M - M - x \\
&\quad \text{(by (7.37), and because there is no hole in } t-1\text{, and } x \text{ tasks are scheduled at } t\text{)} \\
&< (q+1)M - x \qquad (7.39) \\
&\quad \text{(because } t \le t_L,\ \mathsf{LAG}(\tau, t-1) < (q+2)M - 2\alpha M \text{ by (7.38)).}
\end{aligned}$$

We will next express the LAG of τ at t as a sum of the lags of the tasks in τ. Because there is an increase in LAG across slot t, by Lemma 3.4, there is at least one hole in t. Hence, by Lemmas 7.11 and 7.12, the lag of a task in $B(t)$ or $I(t)$ is zero. Therefore, by (3.34), $\mathsf{LAG}(\tau, t+1) = \sum_{T \in A(t)} \mathsf{lag}(T, t+1)$. Further, by Lemma 7.21, there is at least one task T that is scheduled at t with a tardiness of zero, *i.e.*, $T \in A_0(t)$. Hence, by Lemma 7.13, $\mathsf{lag}(T, t+1) < wt(T) \leq W_{\max}$. Because $t \leq t_L < t_d$, the tardiness of every other task scheduled at t is at most q. Hence, by Lemmas 7.14–7.16, $\mathsf{lag}(U, t+1) < (q+1) \cdot wt(U) \leq (q+1)W_{\max}$ holds for every task U scheduled at t other than T. To summarize, of the x tasks scheduled at t, at least one task's lag is less than $W_{\max}$, and those of the remaining tasks are less than $(q+1)W_{\max}$. Hence, $\mathsf{LAG}(\tau, t+1) = \sum_{T \in A(t)} \mathsf{lag}(T, t+1) < (x-1)(q+1)W_{\max} + W_{\max}$. Therefore, because (7.39) also holds, $\mathsf{LAG}(\tau, t+1) < \min((x-1)(q+1)W_{\max} + W_{\max}, (q+1)M - x)$. Because the first expression within this min term is increasing with x while the second is decreasing, LAG is maximized for x obtained by solving $(x-1)(q+1)W_{\max} + W_{\max} = (q+1)M - x$. Solving for x, we have $x = \left(\frac{q+1}{1+(q+1)W_{\max}}\right)M + \frac{q \cdot W_{\max}}{1+(q+1)W_{\max}}$. Substituting x in the first expression within the min expression above, we have

$$
\begin{aligned}
\mathsf{LAG}(\tau, t+1) &< (x-1) \cdot (q+1) \cdot W_{\max} + W_{\max} \\
&= x \cdot (q+1) \cdot W_{\max} - q \cdot W_{\max} \\
&= \left(\frac{(q+1)^2 W_{\max}}{1+(q+1)W_{\max}}\right) \cdot M + \frac{q(q+1)W_{\max}^2}{1+(q+1)W_{\max}} - q \cdot W_{\max} \\
&= \left(\frac{(q+1)^2 W_{\max}}{1+(q+1)W_{\max}}\right) \cdot M - \frac{q \cdot W_{\max}}{1+(q+1)W_{\max}}.
\end{aligned}
$$

If $\mathsf{LAG}(\tau, t+1)$ is at least $f \stackrel{\text{def}}{=} (q+2)M - 2\alpha M + 1$, then the right-hand side of the above inequality exceeds f. This implies that $\alpha M > \frac{((q+1)W_{\max}+(q+2))M+(2q+1)W_{\max}+1}{2(q+1)W_{\max}+2}$, which by (7.37) contradicts (W). The lemma thus follows. ∎

Lemma 7.45 above contradicts the (7.38). Hence, if (W) holds, then there does not exist a time slot across which LAG increases to $(q+2)M - 2\alpha \cdot M + 1$. Thus, our assumption that τ is as defined in Definition 7.8 is incorrect, which establishes Theorem 7.3.

For a given q, the sufficient total utilization determined decreases with M and $W_{\max}$, and it is asymptotically 83.3% for $q = 1$. However, the permissible utilization exceeds 90.0% when $M \leq 10$, and 87.0%, when $W_{\max} = 0.85$. Similarly, for $q = 2$, the asymptotic value is 87.5%, while those for $M = 10$ and $W_{\max} = 0.85$ are 95.0% and 92.99%, respectively. If $M \leq 4$, then the total utilization need not be capped below 100% to ensure that tardiness does not exceed one quantum. It is worth pointing out that deadlines are known to be missed by two quanta on six or more processors.

7.4 Summary

We have presented counterexamples that show that, in general, on M processors, tardiness under the EPDF Pfair algorithm can exceed a small constant number of quanta for task systems feasible on M processors, but whose per-task utilizations are not restricted. Thus, the conjecture that EPDF ensures a tardiness bound of one quantum for all feasible task systems is proved false. We have also presented sufficient per-task utilization restrictions that are more liberal than those previously known for ensuring a tardiness of q quanta under EPDF, where $q \geq 1$. Finally, we have presented a sufficient restriction of $\min(M, \frac{((q+1)W_{\max}+(q+2))M+((2q+1)W_{\max}+1)}{2(q+1)W_{\max}+2})$ on total system utilization for ensuring a tardiness bound of q quanta for use with task systems with $W_{\max} > \frac{q+2}{q+3}$.

Chapter 8

Pfair Scheduling with Non-Integral Task Parameters

In this chapter, we address relaxing Restriction (R2) of Pfair scheduling algorithms. This restriction is specified in Section 3.2 and requires the execution cost and period of each periodic or sporadic task to be specified as integral multiples of the quantum size.[1] In Section 8.1, we consider allowing only periods to be non-integral; handling non-integral execution costs is addressed afterwards in Section 8.2. Finally, in Section 8.3, the impact of non-integral periods in tick-based implementations of non-Pfair scheduling algorithms is briefly considered.

8.1 Pfair Scheduling with Non-Integral Periods

In this section, we consider relaxing a part of Restriction (R2) (specified in Section 3.2), by considering the Pfair scheduling of a periodic or sporadic task system τ with the characteristic that not all tasks of τ have integral periods, but have integral execution costs. We show that τ can be scheduled under PD^2 such that tardiness for any job of τ is less than two quanta. We also show that if τ is scheduled under EPDF, then tardiness for its jobs is worsened by less than two quanta in comparison to the tardiness that can be guaranteed to a task system with the same task weights as those in τ but with integral periods. (It is assumed that $U_{sum}(\tau) \leq M$ holds.)

We make the reasonable assumption that though non-integral, task periods are still positive rational numbers. Therefore, the period $T.p$ of each task T can be specified as a ratio of two

[1] As described in Chapter 3, without loss of generality, a quantum is assumed to be one time unit in duration.

positive integers denoted $T.n$ and $T.m$, such that $T.n$ and $T.m$ are relatively prime. Since $wt(T) = \frac{T.e}{T.p} = \frac{T.e \times T.m}{T.n}$ and $wt(T) \leq 1$, it follows that $T.e \times T.m \leq T.n$. As mentioned above, $T.m$ and $T.n$ are positive integers, and since task execution costs are integral, $T.e$ is a positive integer, as well.

In this chapter, the notation T^k will be used to denote the k^{th} job of T, and hence, subtasks $T_{(k-1)\cdot T.e+1}, \ldots, T_{k \cdot T.e}$ comprise T^k. If T is sporadic, then all subtasks of T^k have the same *offset* or intra-sporadic separation parameter given by the Θ function. We will let $\Theta(T^k)$ denote the *offset of job* T^k, *i.e.*, the amount of time by which T^k is released late in comparison to its release time if T were synchronous, periodic. Therefore $\Theta(T_i) = \Theta(T^k)$, for all $i \in [(k-1) \cdot T.e + 1, k \cdot T.e]$.

As with the Pfair scheduling of task systems for which (R2) holds, we propose scheduling τ by using task weights to assign pseudo-release times and pseudo-deadlines for τ's subtasks. However, if subtask release times and deadlines are assigned according to formulas (3.9) and (3.10), respectively, then relaxing the assumption that not all tasks in τ have integer periods poses the following two complications. (In what follows, let T be a task in τ with a non-integral period.)

(C1) There can exist one or more jobs of T whose deadlines do not correspond to the deadline of any subtask of T. Hence, meeting all subtask deadlines is not sufficient to ensure that all job deadlines of T are met.

(C2) There can exist one or more jobs of T whose release times do not correspond to the release time of any subtask of T.

For an illustration of (C1) and (C2), refer to Figure 8.1. In this figure, T is a synchronous, periodic task with $T.e = 2$, $T.p = \frac{10}{3} = 3.3\bar{3}$, and $wt(T) = \frac{3}{5}$. The PF-windows of the first few subtasks of T are shown in inset (a) and the windows of the first few jobs of T are shown in inset (b). The first job of T, T^1, has a deadline at time $3.3\bar{3}$. This deadline will be met only if T executes for two quanta by $3.3\bar{3}$. However, the pseudo-deadline of the second subtask of T, T_2, is at time 4. Hence, T^1 will miss its deadline unless T_2 is scheduled before time 3, *i.e.*, even if T_2's deadline is met by scheduling it at time 3. Thus, meeting all subtask deadlines is not sufficient to ensure that all job deadlines are met.

(C2) is exemplified by subtask T_3 and the release time assigned to it under regular Pfair scheduling. By (3.9), $\mathsf{r}(T_3) = 3$ ($\Theta(T_3) = 0$, since T is synchronous, periodic). However, since $T.e = 2$, T_3 begins the second job of T, and hence, since $T.p = 3.3\bar{3}$, T_3 is not ready before

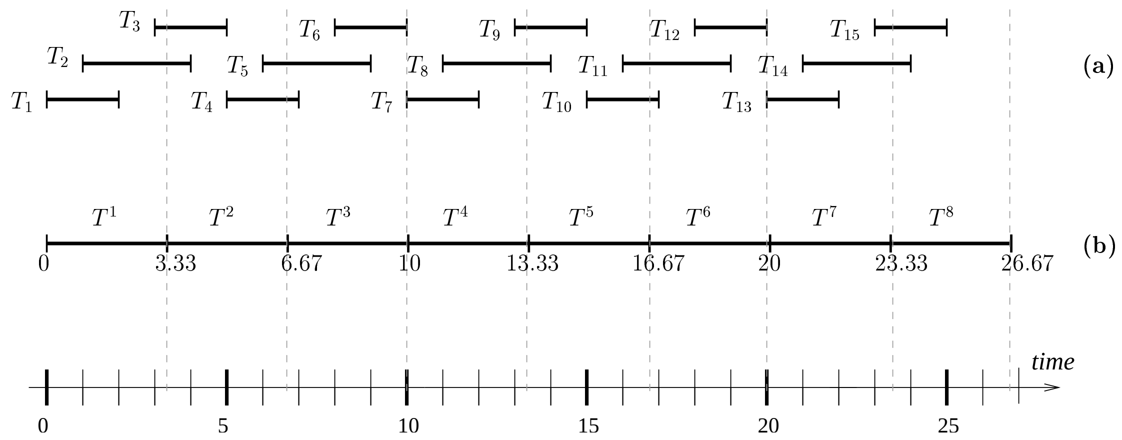


Figure 8.1: T is a synchronous, periodic task with $T.e = 2$, $T.p = \frac{10}{3} = 3.3\bar{3}$, and $wt(T) = \frac{3}{5}$. **(a)** PF-Windows of the first few subtasks of a task with weight $\frac{3}{5}$. **(b)** Windows of the first few jobs of T.

time $3.3\bar{3}$. Therefore, T_3 cannot be scheduled at time 3. The impact of (C2) is illustrated in Figure 8.2(b), which shows a schedule under PD^2 for five tasks each of whose parameters are identical to those of T. In this schedule, two processors are idle at time 3 as the third subtask (*i.e.*, the second job) of no task is ready at that time. This idling leads to several deadline misses in the future.

Lemmas 8.1 and 8.2 below formally quantify the loss to timeliness under PD^2 due to (C1) and (C2), respectively. In Lemma 8.1, we show that for any job, its actual deadline is less than the deadline assigned under regular Pfair scheduling to its final subtask by less than one quantum. It can similarly be shown that for any job, its actual release time is later than the release time assigned under regular Pfair scheduling to its first subtask by less than one quantum. Therefore, the impact of (C2) can be quantified by determining the loss to timeliness when the release times of some needed subtasks of τ are increased by one quantum in comparison to the release times assigned to them under regular Pfair scheduling. For example, for the task system in Figure 8.2, the release time assigned to the third subtask of each task under regular Pfair scheduling is 3; however, since for each task, the third subtask begins the second job whose release time is at 3.33, the release time of the third subtask is increased to 4. The impact of this change is quantified in Lemma 8.2 by showing that if τ is

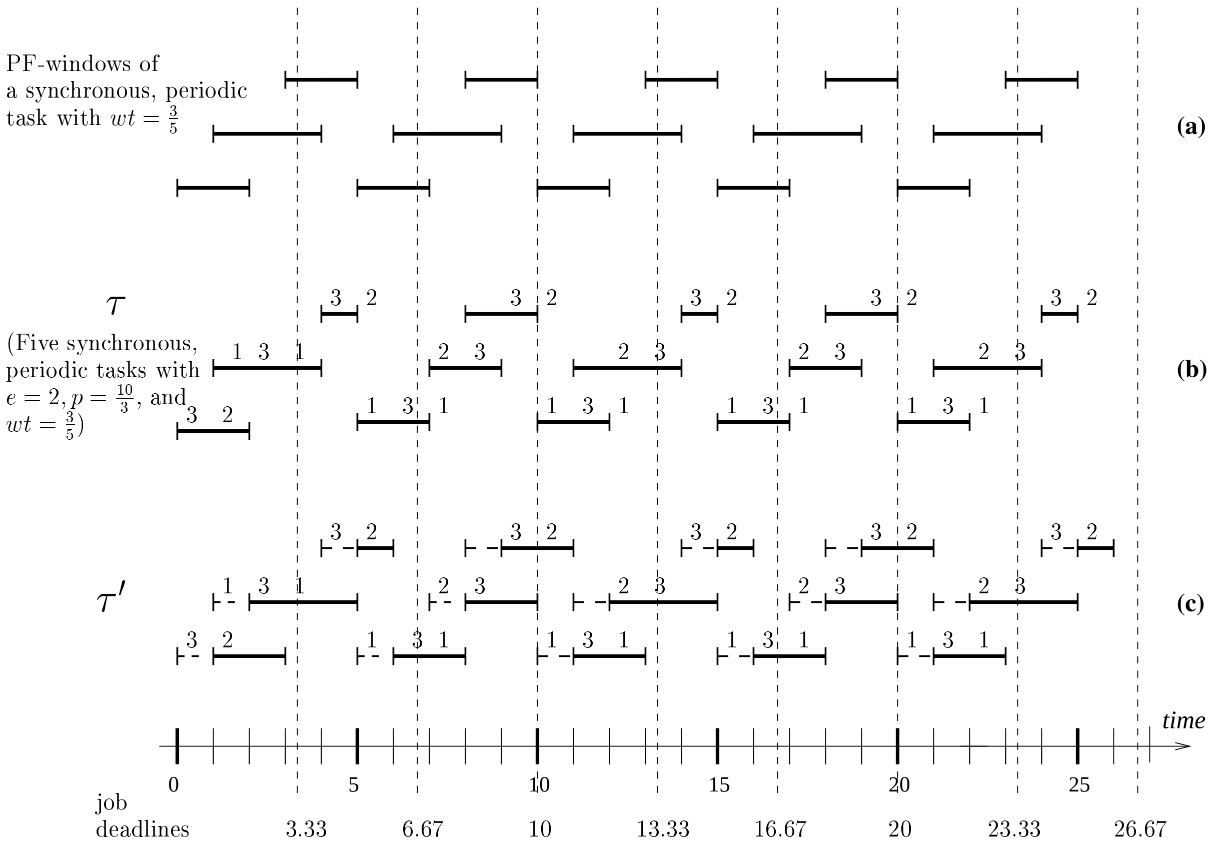


Figure 8.2: Dashed vertical lines denote some initial release times and deadlines of jobs of a synchronous, periodic task with period $= \frac{10}{3}$. **(a)** PF-windows of the first few subtasks of a synchronous, periodic task with weight $\frac{3}{5}$. Release times and deadlines of the subtasks are given by (3.9) and (3.10), respectively. **(b)** A PD^2 schedule on three processors in the interval $[0, 26)$ for a task system τ with five synchronous, periodic tasks with execution cost $= 2$ and period $= \frac{10}{3}$ for each task. Since the weight of each task is $\frac{3}{5}$, the PF-windows for the subtasks of each task under regular Pfair scheduling are as in inset (a). However, the third, fifth, ninth, eleventh, and fifteenth subtasks of each task are not ready in the first slot of their regular PF-windows as these subtasks begin new jobs whose release times are not at slot boundaries. Hence, the release times for these subtasks in τ are increased by one quantum as given by (8.1), and are as shown in this inset. Each subtask window shown represents five instances. An "n" over, or after but aligned with, the i^{th} subtask window indicates that n i^{th} subtasks are scheduled in the corresponding slot. **(c)** An initial segment of a PD^2 schedule for τ' obtained from τ by increasing the release time and deadline of each subtask by one slot. The eligibility time of each subtask is equal to its release time in τ. Note that this schedule is identical to the schedule in inset (b) (assuming that the same "n" tasks as in inset (b) are scheduled in each slot).

scheduled with release times for its subtasks as given by

$$\mathsf{r}(T_i) = \begin{cases} \Theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1, & \frac{i-1}{wt(T)} \text{ is not integral and } i = k \cdot T.e + 1, \text{ for all } k \geq 0 \\ \Theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor, & \text{otherwise} \end{cases} \tag{8.1}$$

then tardiness for the subtasks (not jobs) of τ is at most one quantum. Finally, Lemmas 8.1 and 8.2 are used to show that deadlines are missed by less than two quanta under PD^2 if the restriction that periods be integral is relaxed. For clarity and ease of description, we assume that $\Theta(T_i)$ is integral. It can be shown that eliminating this assumption does not lead to any additional loss to timeliness.

Lemma 8.1 *Let τ be a sporadic task system such that the execution cost of every task in τ is integral. Let T be a task in τ with a non-integral period. Then, for any job T^k of T, $\mathsf{d}(T^k) > \mathsf{d}(T_{k \cdot T.e}) - 1$, where $\mathsf{d}(T_i)$ is as given by (3.10).*

Proof: Since T^k is the k^{th} job of T, its deadline is given by

$$d(T^k) = \Theta(T^k) + k \cdot T.p, \tag{8.2}$$

where, as discussed in the beginning of this section, $\Theta(T^k)$ is the offset of T^k.

As discussed in the beginning of this section again, subtasks $T_{(k-1)\cdot T.e+1}, \ldots, T_{k \cdot T.e}$ comprise the k^{th} job of T, and since T is sporadic, each subtask of T^k has the same offset, which is equal to $\Theta(T^k)$. Specifically, $\Theta(T_{k \cdot T.e}) = \Theta(T^k)$. Therefore, by (3.10), we have

$$d(T_{k \cdot T.e}) = \Theta(T^k) + \left\lceil \frac{k \cdot T.e}{wt(T)} \right\rceil = \Theta(T^k) + \lceil k \cdot T.p \rceil. \tag{8.3}$$

By (8.2) and (8.3), $d(T_{k \cdot T.e}) - d(T^k) < 1$, from which the lemma follows. ■

The next lemma makes use of two task systems that differ only in the release times and deadlines assigned to their subtasks. Hence, in order to identify the task system under consideration, we overload the release time and deadline functions to take a task system as a second parameter. For instance, $\mathsf{d}(T_i, \tau)$ will denote the deadline assigned to a subtask T_i in task system τ.

Lemma 8.2 *Let τ be a sporadic task system with an integral execution cost for every task and a non-integral period for one or more tasks, and let $U_{sum}(\tau) \leq M$. Let the deadline and release time of every subtask T_i of every task T in τ (denoted $\mathsf{d}(T_i, \tau)$ and $\mathsf{r}(T_i, \tau)$) be given by (3.10) and (8.1), respectively. Let the b-bit and the group deadline be as given by (3.6) and (3.7), respectively. Then, tardiness for subtasks of τ is at most one quantum under PD^2, and is worsened by at most one quantum under EPDF, in comparison to the tardiness to which τ may be subject when all task periods are integral but no task weight is altered.*

Proof: Let τ' be a task system obtained from τ such that τ' contains every subtask that is in τ. Let $\mathsf{r}(T_i, \tau')$ and $\mathsf{d}(T_i, \tau')$, the release time and deadline of subtask T_i in τ', be increased by one quantum (in comparison to the values given by (3.9) and (3.10)) to $\Theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1$ and $\Theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil + 1$, respectively. (Note that since $\mathsf{r}(T_i, \tau)$ is given by (8.1) and not (3.9), some subtasks can have equal release times in τ and τ'.) Let the eligibility time of T_i in τ', $\mathsf{e}(T_i, \tau')$, be the same as in τ. Hence,

$$\mathsf{e}(T_i, \tau') = \mathsf{e}(T_i, \tau) \leq \mathsf{r}(T_i, \tau).$$

For instance, if τ is an in Figure 8.2(b), then in τ', the subtasks that correspond to the first few subtasks of τ are as in Figure 8.2(c). Since in τ', all release times and deadlines are uniformly increased by one quantum (in comparison to those assigned under regular Pfair scheduling), and the b-bit and group deadline are not altered for any subtask, the relative priority between any two subtasks is unaltered under PD^2 or EPDF by the increase in release times and deadlines. Hence, since PD^2 is optimal even when subtasks may be early released, it is easy to see that every subtask T_i of τ' completes executing by $d(T_i, \tau')$ when τ' is scheduled under PD^2. Similarly, a tardiness bound of q can be guaranteed to τ' under EPDF, where q denotes the tardiness bound derived in Chapter 7 for a task system with the same maximum task weight as that of τ'.

By the definition of $\mathsf{e}(T_i, \tau')$, the IS-window of every subtask in τ' begins at the same time as in τ, but is extended by one slot to the right (because $\mathsf{d}(T_i, \tau') = \mathsf{d}(T_i, \tau) + 1$ holds for every T_i). Thus, the relative priority between any two subtasks is the same in τ and τ'. Hence, since a subtask remains eligible from its eligibility time until it is scheduled, and the relative priority between any two subtasks is the same in τ and τ', assuming a consistent scheduler that resolves ties identically, the same set of subtasks contend for execution at each slot in corresponding schedules for τ and τ'. Also, the same set of subtasks is scheduled in each slot. Hence, completion times are identical for every subtask in corresponding schedules under the same algorithm (either PD^2 or EPDF) for τ and τ'. For example, initial segments of schedules for τ and τ' under PD^2 depicted in insets (b) and (c) of Figure 8.2 are identical. Finally, as described above, since every subtask T_i of τ' meets its deadline under PD^2, and $\mathsf{d}(T_i, \tau') = \mathsf{d}(T_i) + 1 = \mathsf{d}(T_i, \tau) + 1$ for every T_i, tardiness for τ under PD^2 is at most one quantum. Similarly, tardiness for τ under EPDF is at most $q + 1$ quanta. ■

The theorem below follows from Lemmas 8.1 and 8.2.

Theorem 8.1 *Let τ be a sporadic task system with an integral execution cost for every task and a non-integral period for one or more tasks, and let $U_{sum}(\tau) \leq M$. Then, τ can be scheduled under* PD^2 *such that the tardiness of every job of τ is less than two quanta, and under* EPDF *such that the tardiness of every job of τ is worsened by less than two quanta in comparison to the tardiness to which the job may be subject if all task periods are integral, but no task weight is altered.*

Proof: One way of scheduling τ using Pfair scheduling algorithms is by assigning each subtask T_i of τ a release time that is at least the release time of the job of which T_i is a part, and appropriate values for its deadline and the two tie-break parameters. We propose the following assignments: $\mathsf{d}(T_i)$, $\mathsf{b}(T_i)$, and $\mathsf{D}(T_i)$ are as given by (3.10), (3.6), and (3.7), respectively, and $\mathsf{r}(T_i)$ is as defined in (8.1). As discussed earlier, release times assigned by (8.1) differ from those assigned under regular Pfair scheduling by one quantum for certain subtasks. Refer to Figure 8.2 for an example.

The release time of the k^{th} job of T, T^k, is given by $\mathsf{r}(T^k) = \Theta(T^k) + (k-1) \cdot T.p$. Subtasks $T_{(k-1)\cdot T.e+1}, \ldots, T_{k \cdot T.e}$ comprise the k^{th} job of T, and it can be verified that the release time assigned to the first subtask of T^k using (8.1) is at or after $\mathsf{r}(T^k)$. For example, in Figure 8.2(b), the release time of the third job of each task is at time 6.67. The fifth subtask is the first subtask of the third job and the release time assigned to it is 7. If subtasks of τ are assigned values for the various parameters as proposed above, then by Lemma 8.2, tardiness for the subtasks of τ is at most one quantum under PD^2, and is worsened by at most one quantum under EPDF in comparison to the tardiness bound that can be guaranteed if every task period is integral. By Lemma 8.1, the deadline assigned to the last subtask of a job is less than one quantum later than the actual deadline of the job. The theorem therefore follows. ■

Tightness of the tardiness bound under PD^2. In the schedule in Figure 8.2, the fourth subtask of one of the five tasks does not complete executing until time 8. Since the execution cost of each task is 2.0, it follows that the second job of one of the tasks does not complete executing until that time, whereas its deadline is at time $6.6\bar{6}$. Thus, tardiness for that job is slightly more than 1.33 time units. It is easy to construct examples in which tardiness is arbitrarily close to, but less than, two quanta under PD^2. For example, consider a synchronous, periodic task system τ with $k \cdot e + 1$ tasks with an execution cost of e and a period of $e + \frac{1}{k}$ for each task, where k and e are positive integers. The weight of each task in τ is $\frac{k \cdot e}{k \cdot e + 1}$ and the total utilization of τ is $k \cdot e$. The deadline of the second job of each task is at $2 \cdot e + \frac{2}{k}$, and it is easy to show that the second job (*i.e.*, subtask $2 \cdot e$) of e tasks do not complete executing until time $2 \cdot e + 2$ for a tardiness of $2 - \frac{2}{k}$. This tardiness can be made arbitrarily close to two by choosing a large enough value for k, and therefore, the tardiness bound given by Theorem 8.1 is tight for PD^2.

8.2 Scheduling with Non-Integral Execution Costs

Unfortunately, unlike in the case of task periods, we have not been able to identify an elegant way of eliminating the restriction of Pfair scheduling that task execution costs be integral. However, such tasks can be scheduled in a "Pfair-like" manner by using g-NP-EDF.

To see this, observe that one way of scheduling tasks with non-integral execution costs is by dividing each job of a task into uniform-sized "sub-jobs," allowing different sub-job sizes, if needed, for different tasks, associating a pseudo-release and a pseudo-deadline with each sub-job based on the parameters of its task, and scheduling sub-jobs non-preemptively in an earliest-pseudo-deadline-first basis. There is to be no restriction on job migration in that different sub-jobs of a job can execute on different processors. Since this algorithm is essentially g-NP-EDF applied to sub-jobs, we will refer to it as EDF-sliced, and since g-NP-EDF is not optimal, EDF-sliced is not optimal, either. A tardiness bound that can be guaranteed to a task system τ under EDF-sliced can be computed using the formula provided in Chapter 4 (in Corollary 4.3) by using the highest $M-1$ task utilizations for $\mu_1, \ldots, \mu_{M-1}$, and highest M sub-job execution costs of tasks (as opposed to the highest M task execution costs) for $\epsilon_1, \ldots, \epsilon_M$. Since the tardiness bound of g-NP-EDF given by Corollary 4.3 increases with increasing values for ϵ_i, tardiness can be lowered by choosing smaller sub-job sizes at the expense of increased scheduling and migration overhead.

Though straightforward to compute, for completeness, we provide formulas for the pseudo-release and pseudo-deadline of the i^{th} sub-job of a task (assuming that sub-jobs are numbered sequentially from the first job). At the risk of some notational overload, let T_i denote the i^{th} sub-job of T. Let $T.s$ denote the number of sub-jobs into which each job of T is sliced. Then the pseudo-release $\mathsf{r}(T_i)$ and the pseudo-deadline $\mathsf{d}(T_i)$ of T_i are as follows. (In the formulas below, $\Theta(T_i)$ is the offset of the i^{th} sub-job, and if T is sporadic, then all the sub-jobs of a job have the same offset.)

$$\mathsf{r}(T_i) \quad = \quad \Theta(T_i) + \frac{(i-1) \cdot T.e}{T.s \cdot wt(T)} = \Theta(T_i) + \frac{(i-1) \cdot T.p}{T.s}$$

$$\mathsf{d}(T_i) \quad = \quad \Theta(T_i) + \frac{i \cdot T.e}{T.s \cdot wt(T)} = \Theta(T_i) + \frac{i \cdot T.p}{T.s}$$

For an example application of the above formulas, let T be a synchronous, periodic task with $T.e = 3.2$ and $T.p = 5$, and let each job of T be subdivided into two sub-jobs (*i.e.*, $T.s = 2$). Then, $\frac{T.p}{T.s} = 2.5$, and $\mathsf{r}(T_i) = 2.5 \times (i-1)$ and $\mathsf{d}(T_i) = 2.5 \times i$ (since T is synchronous, periodic, $\Theta(T_i) = 0$). The first sub-job of the first job, T_1, has a deadline of 2.5 and the second sub-job of the same job has a release time of 2.5.

Implementation considerations. Handling non-integral values for sub-job deadlines is similar to handling non-integral periods under g-NP-EDF, which is described in the next section. However, handling non-integral execution costs for sub-jobs cannot be considered to be similar to handling non-integral execution costs for jobs. This is due to the following facts. Assuming a well-behaved system in which tasks do not overrun, job completions need not be policed by the scheduler, and the process thread corresponding to each job either terminates or relinquishes voluntarily when it has executed for its WCET or earlier. In other words, the event that marks the completion of a job is part of the job, upon whose occurrence the scheduler is invoked. On the other hand, sub-jobs are in some sense "artificial," and the execution cost of a sub-job can be viewed as simply defining a point within the job's execution at which its deadline is altered. Further, there is no indication in the job itself that marks the completion of a sub-job, and hence, sub-job completions need to be monitored and enforced using timers. Therefore, practical considerations may require that all sub-jobs except the last of any job have execution costs that are integral multiples of the minimum possible system timer period. This timer period may be equal to the system quantum size in *tick-based* implementations. (Tick-based scheduling is discussed in detail in Chapter 9.) To account for such varying sub-job execution costs, sub-job release times and deadlines given earlier also need to be appropriately modified. As before, letting $T.s$ denote the number of sub-jobs per job of T, the first $T.s-1$ sub-jobs can be assigned an execution cost of $\lceil \frac{T.e}{T.s} \rceil$ (provided $\lceil \frac{T.e}{T.s} \rceil \cdot (T.s-1) < T.e$) or $\lfloor \frac{T.e}{T.s} \rfloor$ each, and the remaining execution cost can be assigned to the final sub-job. Let $T.e_1$ denote the execution cost of each of the initial $T.s - 1$ sub-jobs of a job of T and $T.e_2$ that of the final sub-job. Then, $T.e_2 = T.e - (T.s-1) \cdot T.e_1$, and the release time and deadline of the i^{th} sub-job, T_i, can be computed as follows. Note that because there are $T.s$ sub-jobs per job of T, T_i is part of job $\lceil \frac{i}{T.s} \rceil$. Let I denote the sub-job index of T_i within its job, *i.e.*, $I = ((i-1) \bmod T.s) + 1$.

Then,

$$\begin{aligned}
\mathsf{r}(T_i) &= \Theta(T_i) + \left(\left\lceil \frac{i}{T.s} \right\rceil - 1\right) \cdot T.p + \frac{(I-1) \cdot T.e_1}{wt(T)} \\
\mathsf{d}(T_i) &= \begin{cases} \Theta(T_i) + \frac{i \cdot T.p}{T.s}, & \text{if } T.s \mid i \\ \Theta(T_i) + (\lceil \frac{i}{T.s} \rceil - 1) \cdot T.p + \frac{I \cdot T.e_1}{wt(T)}, & \text{otherwise.} \end{cases}
\end{aligned}$$

For the example task considered earlier (with uniform-sized sub-jobs), letting $T.s = 3$, $T.e_1 = \left\lfloor \frac{T.e}{T.s} \right\rfloor = \left\lfloor \frac{3.2}{3} \right\rfloor = 1$, $T.e_2 = 3.2 - (T.s - 1) \cdot T.e_1 = 1.2$. Since T is synchronous, periodic, all offsets are zero. Hence, $\mathsf{r}(T_1) = 0$, $\mathsf{d}(T_1) = \mathsf{r}(T_2) = \frac{1 \times 1}{3.2/5} = 1.5625$, $\mathsf{d}(T_2) = \frac{2 \times 1}{3.2/5} = 3.125$, $\mathsf{r}(T_3) = 3.125$, and $\mathsf{d}(T_3) = 5$. As discussed in Section 4.4, sub-jobs with varying execution costs can be modeled using the extended sporadic task model described there, and the tardiness bounds derived in Chapter 4 for g-NP-EDF apply.

8.3 Non-Integral Periods under EDF-based Algorithms

Throughout this dissertation, we have assumed that periods and execution costs can be non-integral for non-Pfair algorithms. While a non-integral execution cost is not likely to be problematic in practical implementations, a non-integral period may not be acceptable in tick-based implementations. As mentioned earlier, tick-based scheduling is discussed in detail in Chapter 9, and, under it, new job releases are noticed only at quantum boundaries. Hence, if the period of a task is non-integral, then its jobs may arrive in the middle of a quantum, but will not be considered for scheduling until the next quantum boundary. This is similar to the problem faced under Pfair scheduling when periods are non-integral, which was discussed in Section 8.1. The transformation technique that was used in Lemma 8.2 can be used to show that if practical considerations require periods to be integral, but one or more tasks have non-integral periods, then tardiness under g-EDF, g-NP-EDF, and EDF-sliced is worsened by at most one quantum. (The same technique can be used because the tardiness bounds derived for g-EDF and g-NP-EDF in Chapter 4 apply even if jobs may be early released.) Furthermore, by the same argument, if a task system τ is schedulable without deadline misses under one of these algorithms even if jobs may be early released, then deadlines will be missed by at most one quantum if tasks in τ have non-integral periods but scheduling is tick-based.

Finally, though the tardiness bounds derived for EDF-fm in Chapter 5 cannot be shown to hold if jobs of migrating tasks are arbitrarily early-released, the bounds can be shown to hold if each job is early released by at most one quantum. Hence, the transformation technique of Lemma 8.2 can be used to show that tardiness for fixed tasks is worsened by at most one quantum and that migrating tasks may miss their deadlines by at most one quantum.

8.4 Summary

In this chapter, we considered the Pfair scheduling of tasks with non-integral periods and execution costs. We showed that a task system with non-integral periods for one or more tasks but integral execution costs for all the tasks can be scheduled under PD^2 such that job deadlines are missed by less than two quanta, and under EPDF such that the tardiness bounds guaranteed to its jobs is worsened by less than two quanta in comparison to what can be guaranteed to a task system with comparable task weights but integral task periods. We also proposed an algorithm called EDF-sliced for scheduling tasks in a "Pfair-like" manner when both task periods and execution costs may be non-integral. Finally, we considered the impact to timeliness of non-integral periods in tick-based implementations of non-Pfair scheduling algorithms.

Chapter 9

Performance Evaluation of Scheduling Algorithms

In the previous chapters, we analytically determined some soft real-time guarantees that can be provided by some multiprocessor real-time scheduling algorithms. In this chapter, we attempt to compare the practical performance of the algorithms through simulations. Our metrics for comparison are the percentage of task systems for which each algorithm can guarantee bounded tardiness and the average tardiness bound guaranteed after accounting for system implementation overheads.

The tardiness bound guaranteed by each algorithm considered in this dissertation for a task system is dependent upon the total system utilization, U_{sum}, and holds only if U_{sum} is at most the number of processors. U_{sum} in turn depends on the WCETs and periods of the tasks in τ. As discussed in Chapter 1, system implementation overheads, such as scheduler invocations, and task preemption and migration costs, can take time away from the application tasks at hand and delay them. It was also mentioned that one way of accounting for the time lost due to overheads is to inflate, in an efficient (*i.e.*, non-pessimistic) and a safe manner, the WCETs of tasks above the time required for the execution of each task on a dedicated processor (*i.e.*, in isolation). Further, the extent of the overhead from each extrinsic source can vary with the scheduling algorithm, characteristics of the application, and the implementation platform. Thus, in practice, the tardiness bound guaranteed by an algorithm is dependent on the overheads incurred (because it is dependent on U_{sum}), apart from the intrinsic properties of the algorithm.

Therefore, in this chapter, we seek to empirically determine how the different algorithms compare if external overheads are accounted for. We also include the partitioned-EDF (p-EDF) algorithm in our study. Recall that p-EDF is a no-migration algorithm with EDF as the per-processor scheduler, and as such does not incur any migration overhead. Further, a task system's tardiness under p-EDF is zero if it can be partitioned among the available processors, and unbounded otherwise. Since partitioning algorithms are currently preferred to global algorithms for scheduling real-time multiprocessor systems, we evaluate the other algorithms with respect to p-EDF, which is believed to be among the best partitioning algorithms.

The rest of this chapter is organized as follows. Our assumptions regarding scheduler implementations and task models are stated in Section 9.1. This is followed by a discussion of some significant external sources of overhead in Section 9.2. Then, in Section 9.3, for each algorithm, we give formulas for determining the WCET of a task in the presence of overheads. Finally, a simulation-based evaluation of the algorithms is presented in Section 9.4.

9.1 Assumptions

Before discussing system overheads, we state some notational conventions and our assumptions concerning how tasks are specified and scheduled. As mentioned in Chapter 1, we assume that the underlying hardware platform is a symmetric, shared-memory multiprocessor.

The WCET of a task T on a dedicated processor will be referred to as its *base* WCET and will be denoted $T.e^{(b)}$. The WCET obtained after accounting for overheads will be referred to as its *inflated* WCET, or simply WCET, when unambiguous from context. $T.e$ will refer to T's inflated WCET.

Assumption A1: All task periods are assumed to be expressed as integral multiples of the system's quantum size. However, the base WCET of each task may be non-integral (but a positive, rational number). As shown in Chapter 8, if periods are non-integral, then under all the algorithms considered, tardiness is worsened by comparable amounts that are minimal. Hence, assuming non-integral values for periods does not lead to a loss of generality. Recall that Pfair algorithms require integral values for execution costs as well. This requirement is met by simply rounding up each non-integral inflated WCET. Thus, a non-integral execution cost results in an extra source of overhead for a Pfair algorithm.

Assumption A2: Scheduling is *time driven* (also referred to as *tick scheduling*) under the EPDF and PD^2 Pfair algorithms. As discussed in Chapter 3, quanta are assumed to be aligned on all processors, and scheduling decisions are made only at the beginning of each quantum on each processor. If a Pfair subtask completes executing in the middle of a quantum, then the remainder of that quantum on the associated processor is wasted.

Scheduling under the remaining algorithms is a mix of time-driven and *event-driven scheduling*. As with Pfair algorithms, each processor's scheduler is invoked at the beginning of each quantum to context switch to a higher-priority ready job, if any, and in that sense is time driven. However, unlike Pfair algorithms, a processor's scheduler is also invoked whenever a job completes executing on that processor, even if the completion time is within a quantum (*i.e.*, not at a quantum boundary). In this sense, scheduling is event driven. For illustration, refer to Figure 9.2(a) (considered in detail later), which shows an example schedule under g-NP-EDF. In this schedule, the first jobs of U and V complete execution at time 3.5. Hence, the scheduler is invoked at time 3.5, even though this time is not a quantum boundary, to schedule at most two ready, but waiting, jobs with the highest priorities. As with Pfair algorithms, we assume that quanta are aligned on all processors. Relaxing this assumption will worsen the tardiness bounds guaranteed by the amount by which the quanta are misaligned.

Since periods are assumed to be integral multiples of the quantum size, in a periodic task system, a job becomes ready at a quantum boundary only. Hence, in periodic systems, tardiness is not impacted due to time-driven scheduling. On the other hand, in a sporadic task system, a higher-priority job may arrive within a quantum. In such systems, not scheduling a higher-priority job as soon as it arrives worsens the tardiness of a job by at most one quantum under all of the algorithms, and is thus negligible. Hence, like Assumption A1, this assumption also does not lead to a loss of generality.

It should also be noted that, unlike Pfair algorithms, tick scheduling is not mandatory for non-Pfair algorithms. In implementations that are entirely event driven, the overhead due to tick scheduling is eliminated; the remaining overheads should be similar to those incurred in implementations based on tick scheduling.

Tick scheduling is generally implemented by delivering timer interrupts to each processor at a frequency of $\frac{n}{Q}$ interrupts/sec. (or, equivalently, by using a tick period of $\frac{Q}{n}$), where Q is the quantum size and n is a positive integer. n ticks then comprise a quantum, and a new quantum can be thought of as starting at the beginning of tick $k \cdot n$, for all $k \geq 0$. The scheduler on each processor is invoked at the beginning of each new quantum.

Assumption A3. The amount of data output by a job that is also used by a later job of the same task is minimal. This assumption implies that when a job is scheduled and begins execution for the first time, its working set is not resident on a remote cache in a read-write state. Hence, a new job does not incur overhead involved in invalidating remote cache lines. We believe this to be a reasonable assumption for applications that use time-varying data such as tracking systems. Relaxing this assumption in a non-pessimistic manner requires application-specific details.

Assumption A4. The time taken to load each job's initial working set is included in its base WCET. Because A3 is assumed, this assumption does not cause the base WCET to be dependent on the algorithm.

(Assumptions A3 and A4 will be clearer when preemption and migration overheads are considered.)

Assumption A5. Jobs (or subtasks) are not early released.

9.2 System Overheads

In this section, we discuss several external sources of overhead and the impact of each on the various algorithms. These overheads include *scheduler*, *context-switching*, *cache-related preemption*, and *migration* costs.

Each time a job is preempted, scheduling and context-switching overheads are incurred, and migration overhead is incurred if the preempted job is later scheduled on a different processor. While a job preemption is necessary for a migration, scheduling and context-switching overheads can be incurred even without a preemption, such as, when a lower-priority job is scheduled after a higher-priority job completes execution. Hence, by cache-related preemption overhead, we refer to the overhead due to the eviction from cache of a job's working set, which will be needed later when the job resumes execution on the same processor. This overhead includes neither the scheduling and context-switching costs nor the overhead due to migration.

Before delving into a detailed discussion of the above overheads, we will briefly describe relevant scheduler data structures.

Scheduler data structures. One common approach for implementing periodic tasks is by associating a *release queue* with each needed future quantum. A quantum's release queue is a *priority queue* of all the jobs (or subtasks) that are to be released at the beginning of that quantum. Whenever a job (or subtask) completes executing, if the release time of the next job (or subtask) of the same task is later than the current time, then the next job (or subtask) is enqueued onto the release queue associated with its release time; otherwise, it is enqueued onto the ready queue. Each release queue is merged with the *ready queue*[1] at the queue's release time. Hence, the number of release queues to be maintained at a given time is bounded by the number of tasks. Further, the release time of any non-empty queue is at most $p_{\max}$ quanta in the future, where $p_{\max}$ denotes the maximum period of any task. (Each queue is local to a processor under partitioning and is globally shared otherwise.)

In sporadic systems, job releases are assumed to be triggered by interrupts, either hardware or software, and a job may arrive within a quantum. (Job releases dependent on events external to the system will be delivered using hardware interrupts, whereas those dependent on events in other tasks can be signaled using software interrupts, such as POSIX *signals*.) Each arriving job is queued in the *pending queue*, which is a queue of all jobs arriving within the current quantum (as opposed to a release queue). The pending queue is merged with the ready queue at the beginning of the next quantum. Therefore, there is one logical pending queue under global scheduling, and one pending queue per processor under partitioning (as opposed to multiple release queues).

Under both Pfair and non-Pfair algorithms, the scheduler is invoked at the beginning of each quantum. However, under the non-Pfair algorithms considered here, a change in the set of executing jobs (*i.e.*, jobs that are scheduled on the various processors) may be needed only if some current job completes execution, or some new job (that was not ready at the previous quantum) becomes ready, *i.e.*, the release queue of the next quantum (or pending queue) is non-empty, and the job at the head of the release (or pending) queue has a higher priority than some executing job. Even if no change is needed to the set of executing jobs, housekeeping functions, such as, merging a release (or pending) queue with the ready queue, should be performed at the beginning of each quantum to ensure that the scheduler data structures are updated and consistent. We refer to the time taken at each quantum boundary to perform such housekeeping functions, including the time taken to ensure that no change is needed to

[1] A *ready queue* is a priority queue of ready jobs (or subtasks).

the set of executing jobs (which can vary with the scheduling algorithm), as the *tick-scheduling overhead*, and the time taken to actually effect a change in the set of executing jobs as the *scheduling overhead*.

Tick-scheduling overhead. Let $\mathsf{oh_{tick}}(\mathcal{A})$ denote the worst-case time needed by Algorithm $\mathcal{A}$ in performing housekeeping functions, but not effecting a change in the set of executing jobs, at any quantum boundary. (The housekeeping functions and the tick-scheduling overhead can vary with the scheduler, and are discussed in detail later.)

For non-Pfair algorithms, this overhead is potentially incurred at the beginning of every quantum. That is, effectively at most $Q - \mathsf{oh_{tick}}(\mathcal{A})$ time units are guaranteed to be available within each quantum for executing any job, including the time needed for handling other overheads. Hence, even if a job is allocated an entire quantum, the amount of time that the job actually executes for within that quantum can be lower by $\mathsf{oh_{tick}}(\mathcal{A})$. On the other hand, if a job either commences or resumes execution in the middle of a quantum, then the tick-scheduling overhead is not incurred by the job in that quantum, which is only partially allocated to it. By Assumption A2, a job is not preempted in the middle of a quantum. Hence, for each job, there is at most one quantum, namely, the quantum in which the job completes execution, that is partially allocated to it in which it incurs tick-scheduling overhead. Refer to Figure 9.1 for an illustration. Let the WCET of a task T be e quanta (which may be non-integral), that is, $e \cdot Q$ time units (after accounting for all overheads except that due to tick scheduling). By the above discussion, for non-Pfair algorithms, the number of quanta spanned by any job of T after accounting for the tick-scheduling overhead is given by $\frac{e \cdot Q}{Q - \mathsf{oh_{tick}}(\mathcal{A})}$. Since the number of quanta that are fully allocated to each job of T is at most $\left\lfloor \frac{e \cdot Q}{Q - \mathsf{oh_{tick}}(\mathcal{A})} \right\rfloor$ and each job may incur tick-scheduling overhead in at most one quantum that is partially allocated to it, the number of tick-scheduling costs to charge for each job of T is at most $\left\lceil \frac{e \cdot Q}{Q - \mathsf{oh_{tick}}(\mathcal{A})} \right\rceil$. Hence, the overhead due to tick scheduling is at most $\left\lceil \frac{e \cdot Q}{Q - \mathsf{oh_{tick}}(\mathcal{A})} \right\rceil \cdot \mathsf{oh_{tick}}(\mathcal{A})$ time units. By adding this overhead to $e \cdot Q$ (the WCET that includes all overheads except that due to tick scheduling), the overall inflated execution cost of T is at most $e \cdot Q + \left\lceil \frac{e \cdot Q}{Q - \mathsf{oh_{tick}}(\mathcal{A})} \right\rceil \cdot \mathsf{oh_{tick}}(\mathcal{A})$ time units, that is, $e + \left\lceil \frac{e \cdot Q}{Q - \mathsf{oh_{tick}}(\mathcal{A})} \right\rceil \cdot \frac{\mathsf{oh_{tick}}(\mathcal{A})}{Q}$ quanta.

For Pfair algorithms, the same set of functions is performed at each quantum boundary regardless of whether the set of executing jobs changes; hence, the tick-scheduling overhead is subsumed in the scheduling overhead.

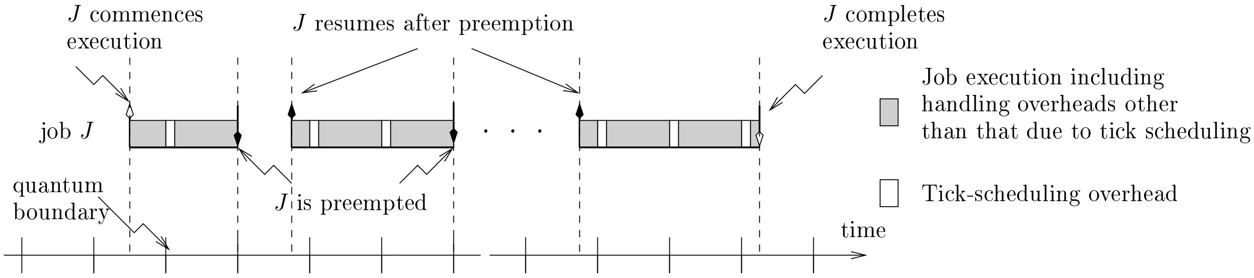


Figure 9.1: Accounting for tick-scheduling overhead in a non-Pfair algorithm. Note that a job is preempted only at quantum boundaries, whereas it may commence, resume, or complete execution within a quantum.

Scheduling overhead. *Scheduling overhead* refers to the time taken in effecting a change to the set of jobs executing on a processor, and, like the tick-scheduling overhead, is dependent on the scheduling algorithm.

The tick-scheduling and scheduling overheads consist of one or more of the following: *enqueuing overhead*, *merging overhead*, and *dequeuing overhead*. Each of these overheads is described in turn below.

Enqueuing overhead refers to the time taken to enqueue a preempted job onto the ready queue, or, in the case of periodic task systems, the next job (or subtask) of a task (whose job or subtask completes executing) onto a release queue or the ready queue. In sporadic systems, enqueuing overhead includes the time needed to enqueue a newly-arriving job onto the pending queue.

It is immediate that in periodic task systems, enqueuing overhead is incurred only when a change is needed to the set of executing jobs, and hence, is a part of the scheduling overhead. On the other hand, for sporadic task systems, though the arrival of one or more new jobs within a quantum implies that one or more jobs will be ready at the next quantum boundary, necessitating a merge of the pending queue with the ready queue, a change to the set of executing jobs may not be warranted. In other words, in sporadic task systems, enqueuing overhead can be incurred even at quantum boundaries at which there is no change to the set of executing jobs. Therefore, it is not appropriate to consider the enqueuing overhead to be part of the scheduling overhead; it should rather be considered to be part of the tick-scheduling overhead. Furthermore, the magnitude of the enqueuing overhead can be higher for sporadic task systems due to the following reasons. First, in a periodic system, at most one job is enqueued per change in the set of executing jobs. On the other hand, in a sporadic system, potentially $\mathcal{O}(N)$ jobs could arrive at or before a quantum boundary all of which need to be enqueued.[2] Secondly, in a sporadic system, each job arrival can require the interrupt service routine (ISR) or the signal handler of the job-arrival interrupt to be invoked, and hence, enqueues are only logically but not physically part of the scheduler. On the other hand, in a periodic system, the cost of invoking a separate ISR before an enqueue operation can be eliminated because the single enqueue can be physically part of the scheduler.

Merging overhead refers to the time taken at a quantum boundary to merge the next quantum's release (pending) queue with the ready queue in periodic (sporadic) systems. As discussed earlier (in the context of scheduler data structures), a merge operation may potentially be required at each quantum boundary, and hence, is a part of the tick-scheduling overhead.

Finally, dequeuing overhead refers to the time required to extract or dequeue the highest priority job from the ready queue, and is a part of the scheduling overhead.

The time complexities of each of the above overheads, and hence, those of tick-scheduling and scheduling, depend on the size of the ready queue (from which the highest priority job is chosen) and the sizes of the release and pending queues and whether a processor's queues are exclusive to it or shared. Since each task can have at most one ready job (or subtask) at any time (even though more than one is pending), for global algorithms, the size of the global ready queue is at most N. Under p-EDF and EDF-fm, the number of tasks assigned to a processor is $N - (M - 1)$ in the worst-case (assuming at least one task is assigned to each processor), and $\frac{N}{M}$ on an average. (It should be noted that in sporadic systems, additional mechanisms are needed to ensure that multiple pending jobs of a task are not present in the ready queue. Otherwise, the size of the ready queue will depend on the number of jobs that can be pending per task, which in turn depends on the tardiness bound. One mechanism is to associate a pending queue per task, as opposed to a single common pending queue for all the tasks, and enqueue each arriving job onto the pending queue of its task in a FIFO manner. At each quantum boundary, only jobs at the head of each pending queue, and whose predecessors have completed execution, should be merged with the ready queue.) In a periodic system, because a job is enqueued onto a release queue only after its predecessor completes executing, each task can have at most one job in all the release queues taken together. Hence, the size of

[2] Recall that in a periodic system, a new job is enqueued (either in a release queue or in the ready queue) when the prior job of the same task completes. Hence, though $\mathcal{O}(N)$ jobs can have their release times at some quantum boundary, such as the hyperperiod, this does not imply that $\mathcal{O}(N)$ jobs are enqueued at that time. The number of jobs enqueued at any time, including the hyperperiod, is given by the number of jobs completing execution at that time, which is at most M.

a release queue has the same upper bound as that of the ready queue. Similarly, in a sporadic system, since at most one job per task will be merged with the ready queue at any quantum, the size of the pending queue can be taken to be at most N in determining the time complexity of a merge.

Digression: Implementation of EDF-fm. Before continuing with the discussion of time complexity, a brief description of how EDF-fm can be implemented is in order. Recall that under EDF-fm, each processor is assigned some number of fixed tasks and at most two migrating tasks, and a fixed task is assigned to a single processor, while a migrating task is assigned to two processors. Also recall that each job of a migrating task executes on one of its two processors only. Further, the processor to which a migrating task's job is assigned can be determined based on the job index. Hence, if a migrating task is periodic, each of its processor schedulers can easily determine the release time of the next job to be assigned to it, and can enqueue it in the appropriate release queue when the current job of that task completes execution. (Such an approach is sufficient since migrating tasks have zero tardiness.) On the other hand, if a migrating task is sporadic, then each of its job arrivals can be programmed to trigger interrupts on both of its processors. Here again, since the processor on which the job executes depends on the job index, one of the processors can simply ignore arrival interrupts that are not meant for it. Further, the two logical ready queues (one each for the ready jobs of fixed and migrating tasks) associated with each processor can be implemented using a single physical ready queue with a bit to denote whether a job belongs to a fixed task. Thus, implementing EDF-fm incurs very little overhead, if any, in comparison to p-EDF.

Scheduler time complexity. We now turn to determining the worst-case time complexity of the various priority-queue operations, and the total complexity of the sequence of operations associated with scheduling and tick-scheduling. We will first ignore synchronization costs incurred under global algorithms and include them later. Assuming that binomial heaps[3] [117] are used for implementing the ready queue and release queues, from our earlier discussion on queue sizes, it follows that the worst-case cost of one enqueue, dequeue, or merge operation is $\mathcal{O}(\lg N)$ under all the algorithms for both periodic and sporadic systems. Hence, for periodic systems, the worst-case complexity of both tick-scheduling and scheduling on any processor

[3]The worst-case time complexity of merging two binomial heaps is $\mathcal{O}(\lg n)$ whereas it is $\mathcal{O}(n)$ for binary heaps, where n is the total number of elements in the two heaps. Further, binomial heaps are not more complicated than binary heaps to implement.

in $\mathcal{O}(\lg N)$. On the other hand, for sporadic systems, while the worst-case complexity of scheduling is $\mathcal{O}(N)$, that of tick-scheduling is $\mathcal{O}(N + \lg N)$. This is because, $\mathcal{O}(N)$ jobs (or subtasks) can arrive within a quantum, or be at the heads of the pending queues of tasks, waiting to be merged. Constructing a binomial heap for the pending jobs that need to be merged requires $\mathcal{O}(N)$ time, while merging the constructed heap with the ready queue requires $\mathcal{O}(\lg N)$ time. If tasks are partitioned approximately evenly among processors, then the complexity of scheduling under p-EDF and EDF-fm can be lowered to $\mathcal{O}(\lg \frac{N}{M})$ for periodic systems, and $\mathcal{O}(\frac{N}{M} + \lg \frac{N}{M})$ for sporadic systems.

We now consider the overhead incurred by global scheduling algorithms due to the use of shared, global queues. Apart from having to deal with larger queue sizes, global algorithms also suffer from the drawback that accesses to the shared queues must be serialized. Hence, in the worst case, any processor waits for every other processor to complete its scheduling or tick-scheduling operations. Thus, for both periodic and sporadic task systems, the complexity of scheduling under global algorithms is $\mathcal{O}(M \lg N)$. Since "merging" is the predominant source of tick-scheduling overhead, and at most one processor needs to perform a merge for any quantum, for periodic systems, the complexity of tick scheduling on each processor is $\mathcal{O}(M + \lg N)$ (and not $\mathcal{O}(M + M \lg N)$). The term M in $\mathcal{O}(M + \lg N)$ accounts for the constant-time overhead incurred on each processor to ensure that the merge has completed. Similarly, for sporadic systems, the complexity of tick-scheduling overhead per processor is $\mathcal{O}(M + N + \lg N)$ (and not $\mathcal{O}(M + MN + M \lg N)$).

Context-switching overhead. Context-switching overhead refers to the actual cost of switching between two tasks and does not include any task-specific cache-related overheads. Overhead due to context switches primarily includes the time needed to save the hardware context of the preempted or completing job in its process control block (PCB), load into cache the PCB of the newly-scheduled job, and populate needed hardware registers from the loaded PCB. If tasks do not share a common address space, then some additional overhead may be incurred to invalidate and repopulate the translation look-aside buffer (TLB), and to load into cache the page-table entries of the task that is being switched to. The cost of this overhead should be nearly equal for all the algorithms.

Cache-related preemption overhead. This overhead refers to the delay associated with servicing the cache misses suffered by a preempted job (misses that would have been hits but for the preemption) when it resumes execution on the same processor (that it executed upon before preemption). Preemption overhead[4] is maximized when the tasks executing on all the processors load their caches at the same time. Further, the delay depends on the amount of data and instructions being fetched. Hence, the worst-case preemption cost is dependent on task characteristics such as the size of its working set and its code size, and is independent of the scheduling algorithm. Though it may be argued that this overhead can be incurred by a newly-scheduled job also, and not just a job resuming execution after a preemption, we assume that Assumption A4 holds.

Migration overhead. This overhead refers to the delay associated with servicing the cache misses suffered by a preempted job when it resumes execution on a different processor. Even though a preempted job is assumed in this case to resume execution on a different processor, the worst-case migration cost is the same as the worst-case preemption cost if the data being loaded is not resident on another processor's cache (specifically, the processor on which the job executed before the preemption) in a conflicting state (*i.e.*, read-only versus read-write or read-write versus read-write states). Otherwise, additional delay may be incurred in invalidating appropriate cache lines in the cache of the first processor. (As with preemption overhead, the worst case is when data is moved from a remote cache to the local cache on *each* processor, though such a scenario seems highly unlikely.) Hence, since code segments are read-only, migration costs are the same as preemption costs for instructions. (Note that an un-flushed cache constitutes a worst-case scenario if a job migrates and a best-case scenario if the job resumes on the same processor.) Finally, because jobs do not migrate under p-EDF, EDF-fm, or NP-EDF, by Assumptions A3 and A4, migration overhead is zero for these three algorithms. For the remaining algorithms, it is dependent on the working set size of the migrating task. Hence, for a given task, the worst-case value should be identical for all unrestricted-migration algorithms.

The worst-case overhead incurred by Algorithm $\mathcal{A}$ due to each of the above factors will be denoted using the following notation. Since cache-related preemption and migration overheads are dependent on task characteristics, these overheads take a task as a second parameter.

$$\mathsf{oh}_{\mathsf{tick}}(\mathcal{A}) \stackrel{\text{def}}{=} \text{overhead due to tick scheduling}$$

[4]When unambiguous, we will refer to cache-related preemption overhead as just preemption overhead.

$$\begin{aligned}
\mathsf{oh}_{\mathsf{sch}}(\mathcal{A}) &\stackrel{\text{def}}{=} \text{scheduling overhead} \\
\mathsf{oh}_{\mathsf{cs}}(\mathcal{A}) &\stackrel{\text{def}}{=} \text{context-switching overhead} \\
\mathsf{oh}_{\mathsf{pr}}(\mathcal{A}, T) &\stackrel{\text{def}}{=} \text{preemption overhead} \\
\mathsf{oh}_{\mathsf{mig}}(\mathcal{A}, T) &\stackrel{\text{def}}{=} \text{migration overhead}
\end{aligned}$$

9.3 Accounting for Overheads

In this section, we show how to account for the overheads discussed in Section 9.2, except the overhead due to tick scheduling, under each algorithm considered. For non-Pfair algorithms, the tick-scheduling overhead $\mathsf{oh}_{\mathsf{tick}}(\mathcal{A})$ should be incorporated using the formula provided in Section 9.2 after the remaining overheads are accounted for as described below. We do discuss elements that contribute to $\mathsf{oh}_{\mathsf{tick}}(\mathcal{A})$ (apart from the merging overhead discussed earlier) and how to determine this value for each algorithm. For Pfair algorithms, as mentioned earlier, the tick-scheduling overhead is included in the scheduling overhead. Our approach to accounting for all the overheads considered is similar to that commonly used in work on real-time systems. We will begin with g-NP-EDF.

Algorithm g-NP-EDF. Since there are no preemptions under g-NP-EDF, each job is scheduled exactly once. Thus, the total number of scheduling decisions (*i.e.*, the total number of dequeue operations in which the highest priority job is dequeued from the ready queue) needed to effect changes in the set of executing jobs is equal to the total number of jobs. Similarly, the total number of context switches is equal to the total number of jobs. Hence, overhead due to scheduling and context switches is fully accounted for under g-NP-EDF if each job is charged with one worst-case scheduling cost and one worst-case context-switching cost (to account for scheduling that job and context switching to it). Refer to Figure 9.2 for an example. Inset (a) shows an NP-EDF schedule in which overheads are assumed to be zero. If this assumption does not hold, then job executions may be delayed due to scheduling and context switching costs as shown in inset (b). Since there are no job preemptions or migrations, by Assumption A3, cache-related preemption and migration costs are zero. Therefore, overhead under g-NP-EDF (excluding that due to tick scheduling) can be fully accounted for if the WCET of each task T is determined as follows. (Recall that $T.e$ and $T.e^{(b)}$ denote T's inflated and base WCETs,

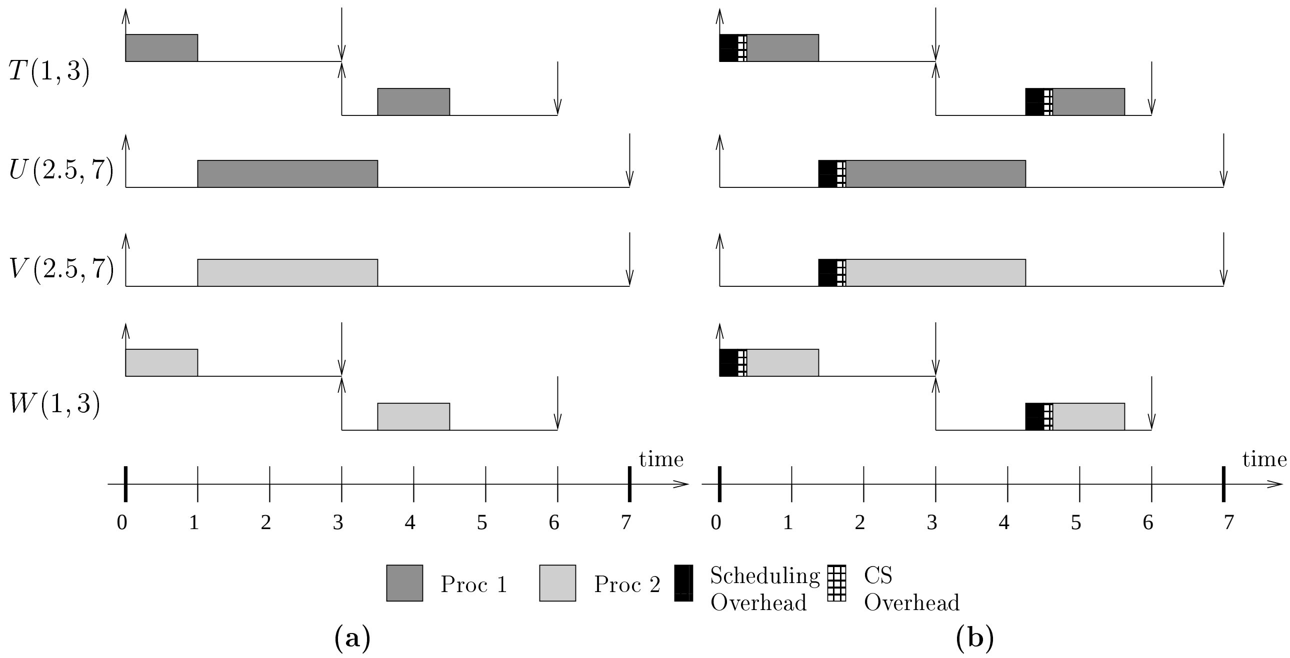


Figure 9.2: Example g-NP-EDF schedules with **(a)** zero overhead and **(b)** non-zero overhead.

respectively.)

$$T.e = T.e^{(b)} + \mathsf{oh}_{\mathsf{sch}}(\text{g-NP-EDF}) + \mathsf{oh}_{\mathsf{cs}}(\text{g-NP-EDF})$$

Under g-NP-EDF, tick-scheduling overhead consists of the time taken at the beginning of a quantum to merge that quantum's release queue or the pending queue with the ready queue. Although this overhead is incurred on at most one processor, since tasks are not bound to processors, the worst case is to assume that the overhead is incurred on every processor, and to charge each task.

Algorithm g-EDF. g-EDF differs from g-NP-EDF in that jobs may be preempted. Hence, overhead under g-EDF is higher than that under g-NP-EDF by the overhead induced by preemptions.

To see how to account for all the overheads induced by preemptions (including additional scheduling and context switching overheads), consider an arbitrary preemption in which J_2 is the preempted job. Consider the scenario when J_2 resumes execution after this preemption. Since g-EDF is work conserving, there exists a unique job J_1 with higher priority than J_2 such that J_2 resumes execution immediately after J_1's completion (on the processor on

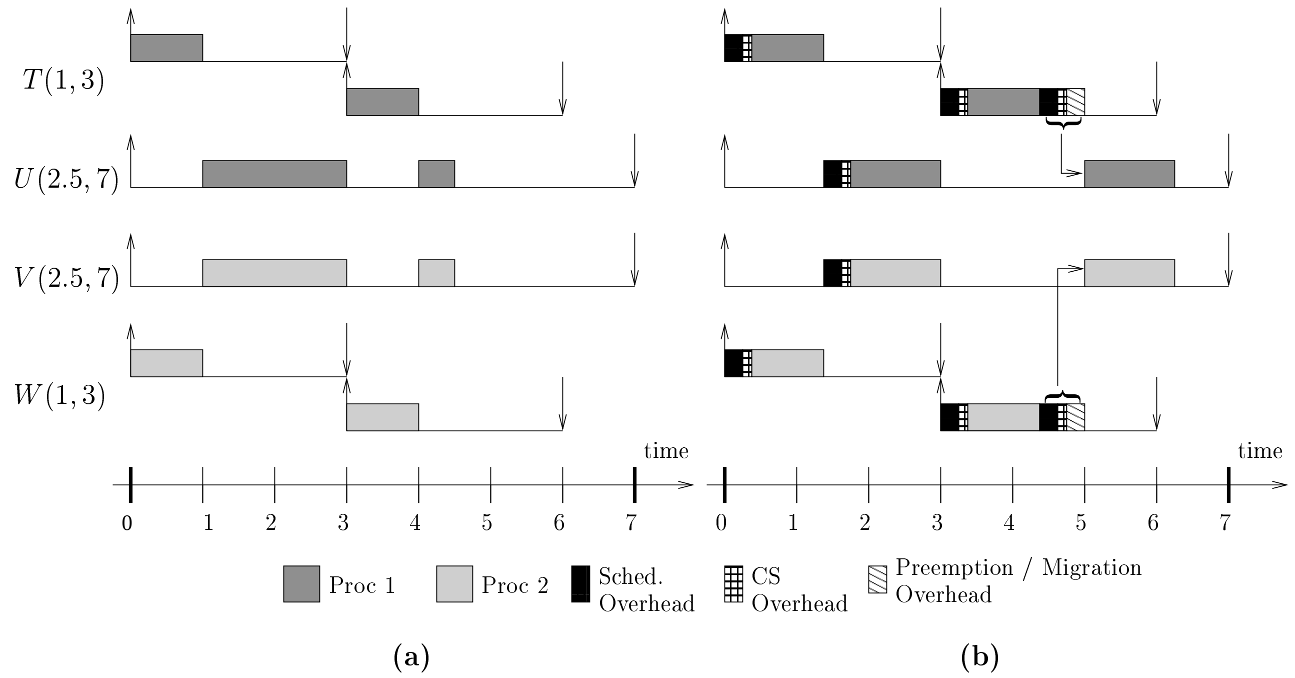


Figure 9.3: Example g-EDF schedules with **(a)** zero overhead and **(b)** non-zero overhead.

which J_1 completes). Therefore, all the overhead incurred in resuming J_2, namely scheduling, context-switching, and either cache-related preemption or migration overhead (depending on the processor on which J_2 executed before being preempted), as discussed in Section 9.2, can be charged to J_1. (Since the worst-case migration cost is at least equal to the worst-case cache-related preemption cost, it is conservative to charge J_1 with J_2's migration cost regardless of whether J_2 migrates.) Since J_1 completes execution exactly once, at most one preempted lower-priority job can resume execution after J_1 completes, and hence, it suffices to charge J_1 with the overhead incurred in resuming at most one lower-priority job. Refer to Figure 9.3 for an illustration. In this example, jobs of U and V resume after the second jobs of T and W, respectively, complete execution, and the completing jobs are charged with the overhead incurred in resuming the preempted jobs. Therefore, because each preempted job resumes execution after some higher-priority job completes execution, it follows that all the overheads due to preemptions are fully accounted for if each job J is charged with the overhead of resuming exactly one lower-priority job in addition to the overhead incurred when J is scheduled and executed for the first time (which is the same as that under g-NP-EDF).

It should be mentioned that though the accounting mechanism described above will never underestimate the actual overhead incurred, it can be a little too conservative and overestimate.

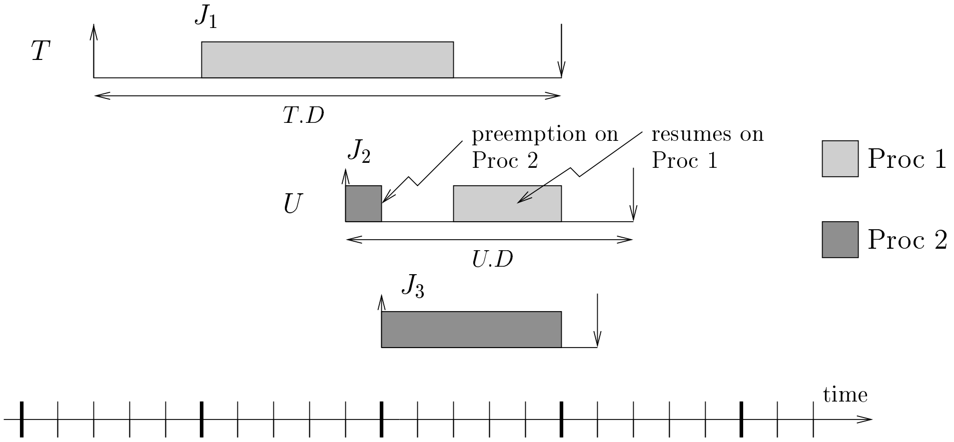


Figure 9.4: Example to illustrate that if a job of task U resumes after a job of task T completes, then $U.D > T.D$ need not hold.

For instance, in a soft real-time system, if some job is tardy, then it is very likely that its successor job or a job that has not yet commenced execution is scheduled when this tardy job completes, as opposed to a preempted job of a different task. In such a scenario, the completing job incurs no overhead associated with resuming a preempted job. Similarly, since the total utilization of a multiprocessor-based hard real-time system scheduled under g-EDF will be much less than 100%, the number of preemptions is likely to be much fewer than that required for each completing job to resume a preempted job. Thus, charging each job with the overhead incurred in resuming one lower priority job can be overkill. However, it is not straightforward to improve the accounting mechanism without guaranteeing that tasks will never be undercharged. We defer further investigation on this problem to future work.

Of all the overheads charged to a job of a task T, only the cache-related preemption and migration costs are task dependent. We will now determine an upper bound for these costs. Note that these costs depend on each task U whose jobs may resume execution after any job of T. On a uniprocessor, it can be shown that the relative deadline of each such task U, $U.D$, is larger than that of T (because, by Assumption A5, there are no early releases). However, the same does not hold for multiprocessors. For this, consider the example in Figure 9.4. In this example, job J_2 of task U resumes immediately after job J_1 of task T completes execution. However, $T.D > U.D$. Thus, since no relation can be established between T and U, the worst case is to assume that a job of a task with the largest migration overhead resumes execution after a job of T. Hence, for all T (in the absence of other task-system-specific information), it is necessary to charge a value of $\max_{U \in \tau \wedge U \neq T} \mathsf{oh}_{\mathsf{mig}}(\mathsf{g\text{-}EDF}, U)$ to fully account for cache-related overheads incurred with a preemption.

To summarize, under g-EDF, all system overheads (except that due to tick scheduling) can be fully accounted for by inflating the WCET of each job of each task by the worst-case costs of two scheduling decisions, two context switches, and the cost of refetching some data and instructions evicted from the cache for one lower priority job. By the discussion above, the following formula can be used to determine an inflated WCET for task T.

$$T.e = T.e^{(b)} + 2 \cdot \mathsf{oh}_{\mathsf{sch}}(\mathsf{g\text{-}EDF}) + 2 \cdot \mathsf{oh}_{\mathsf{cs}}(\mathsf{g\text{-}EDF}) + \max_{\{U \in \tau \wedge U \neq T\}}(\mathsf{oh}_{\mathsf{mig}}(\mathsf{g\text{-}EDF}, U))$$

Tick-scheduling under g-EDF. In what follows, we describe some complexities in ensuring that a job whose execution spans multiple, contiguous quanta is scheduled on the same processor, and is not migrated needlessly, when a change is needed to the set of executing jobs at a quantum boundary. Such complexities necessitate additional operations apart from a queue merge at quantum boundaries, and hence, the overhead due to tick scheduling is likely to be higher under g-EDF than under g-NP-EDF.[5] We will use the g-EDF schedule in Figure 9.5 to illustrate the complexities involved and some possible solutions. In this schedule, delays due to tick scheduling are not depicted; the schedule is rather used to illustrate why additional operations will be needed at some quanta. We will begin by considering some simple scenarios and then move to complex ones.

In Figure 9.5, at time 1 or time 2, no new job is released. Hence, at these times, the jobs that are executing can be resumed after brief interruptions incurred in switching to kernel mode, verifying the status of the quantum's release queue,[6] and returning to user mode; thus, the overhead due to tick-scheduling is limited to these operations. At time 21, task W's second job, W^2, is released. Therefore, the tick-scheduling activity for time 21 requires that the kernel on one of the processors merge the release queue with the ready queue (in addition to every processor switching to kernel mode and testing the status of the release queue). However, the deadline of W^2 is at time 42, and hence, W^2, which is the only job in the release queue for time 21 (and is also the job at the head of that release queue), has lower priority than each

---

[5]It should be mentioned that some additional operations will be necessary under g-EDF even if contiguous job executions need not be scheduled on the same processor.

[6]More accurately, since the first processor that finds a non-empty release queue will merge it with the ready queue, when a processor finds an empty release queue, it must verify that no new job with higher priority is at the head of the ready queue. For ease of description, we will refer to this operation as verifying the status of the release queue.

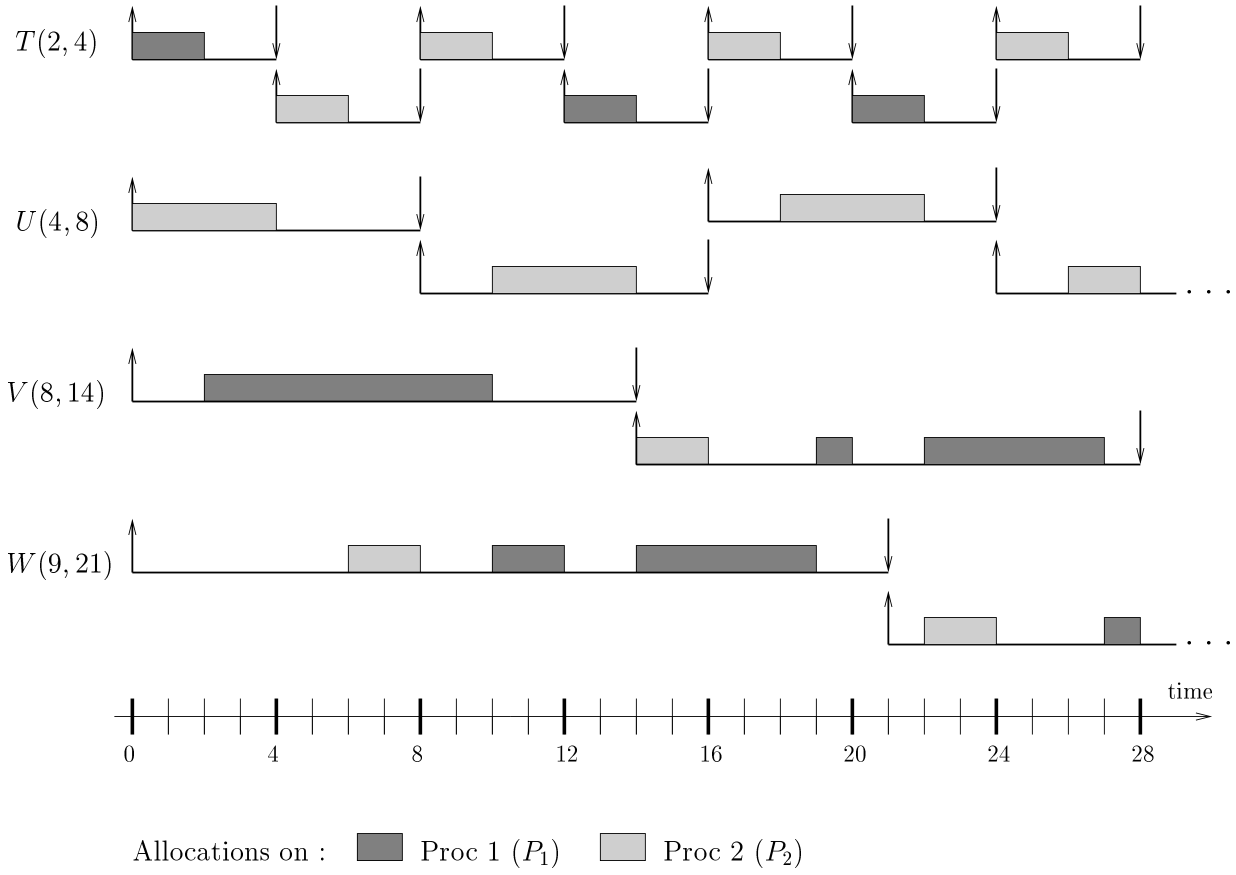


Figure 9.5: Example g-EDF schedule to illustrate some complexities in ensuring that a job whose execution spans contiguous quanta is not migrated needlessly.

of the executing jobs and the job at the head of the ready queue (which is V^2 with deadline at time 28). Therefore, in comparison to g-NP-EDF, the only additional operations (over a queue merge) needed at this time are for each processor to compare the priority of its *current job* (*i.e.*, the job executing at a quantum boundary when the occurrence of a tick interrupt invokes the scheduler) to that of the job at the head of the release queue, and to resume its current job. Thus, the overhead incurred here is nearly the same as the worst-case overhead incurred under g-NP-EDF.

Moving to more complex scenarios, if at a quantum boundary, some newly-released job has higher priority than some current job, then the priority comparison described above will not suffice: selecting the set of jobs to be scheduled in the following quantum may require making M scheduling decisions even if the execution of some jobs will span contiguous quanta (*i.e.*, even if not every current job will be preempted) before a processor decides whether its current job needs preemption. This is because if a newly-released job has higher priority than some current job, then a processor whose current job's priority is lower cannot *independently* determine through a single priority comparison how many of the currently executing jobs have higher priority than its job and whether its job should continue.[7] To ensure that jobs are not needlessly migrated, in the worst case, each processor must wait until M highest priority jobs have been determined before identifying the job to execute on it in the next quantum. Also, housekeeping operations associated with each current job and its task should have been completed by its processor's scheduler before job selections for the next quantum can commence. Such operations include updating the job's status to pending or complete, and if the job is complete, enqueuing the task's next job in the ready or the release queue.

For an illustration of the above aspects, consider times 4 and 20 in Figure 9.5, when T^2 and T^6, respectively, are released with higher priorities than any executing or ready job. If scheduling is centralized (distributed scheduling is considered afterwards), then regardless of which processor is the central scheduler, at time 4, the execution of V^1 has to be temporarily stalled on processor P_1 until $M = 2$ scheduling decisions have been made to conclude that V^1 has the second highest priority, and hence, can continue execution. Similarly, at time 20, the executions of both U^3 and V^2 should be stalled and $M = 2$ highest priority jobs identified before deciding to resume U^3 and preempt V^2. Also, as mentioned above, before determining the jobs to be scheduled, the central scheduler should wait until the remaining processors have updated the status of their current jobs, and have completed enqueue operations for next jobs of tasks of completing jobs, if any. For instance, if P_1 is the central processor, then at time 4, unless P_2 has updated the status of job U^1 as complete and enqueued its next job, the scheduling decisions of P_1 may be incorrect (in that U^1 may be included in the set of jobs for time 4).[8] If scheduling is distributed, wherein each processor selects one highest priority job, then each processor should wait for all the other processors to complete making their selections so that any job whose execution is contiguous can be correctly assigned. In the example, at time 4, if P_1 is the first processor to make a scheduling decision and selects the job with the highest priority, and hence, chooses T^2, then unless P_1 waits for P_2 to select V^1 (which is the job with the second highest priority), and does not dispatch T^2 prematurely, V^1 can migrate to

[7]Of course, a processor whose current job's priority is not lower than that of the job at the head of the release queue can safely continue executing its current job.

[8]The alternative of entrusting the central processor with the responsibility of updating the status of jobs running on other processors is likely to be cumbersome, especially when a job executes for less than its WCET, or when a job commences or resumes execution within a quantum (and not at a boundary) on a remote processor.

P_2. Furthermore, as with the centralized case, each processor should wait for other processors to complete updating the status of their current jobs before making its selection, and thus, tick scheduling may require two passes. Finally, mechanisms will be needed to ensure that the current jobs are considered efficiently along with those in the ready queue while making scheduling decisions. For instance, consider time 20, when both the executing jobs have to be considered along with T^6, the job merged onto the ready queue, in deciding which jobs run during $[20, 21)$.

g-EDF implementation. We now describe one possible approach, which is distributed, for efficiently ensuring that all possible complexities are handled and needless job migrations are eliminated. We propose ordering the processors such that each processor can make just one selection, and yet, dispatch the job it selects without waiting for the remaining processors to complete their selections. (In other words, the proposed processor order can ensure that each current job that will execute in the next quantum is selected by its current processor.) This processor order is given by the reverse priority order of the processors' current jobs that will not complete by the end of the current quantum (and hence will be eligible and contend for execution in the next quantum)[9] and is maintained in a *reverse priority queue* called a *running queue*. A processor that is idle, or whose current job *will* complete by the end of the current quantum, will have a *dummy* or placeholder job with the lowest priority to represent it in the running queue.

Given this running queue data structure, while making selections at the next quantum boundary, each processor should wait for its dummy or current job to be at the head of the running queue. If it has a dummy job at the head, then the processor simply selects the job (if any)[10] at the head of the ready queue for execution on it; else, the processor selects either its current job or the job at the head of the ready queue, depending on which job has higher priority, resolving any tie in favor of its current job. In either case, the processor deletes its entry (current or dummy job) from the current running queue, and enqueues its new entry in the running queue for the next quantum. Since a processor that is idle or whose current job will not be pending at the next quantum has no constraints on selecting a newly-released job, letting such a processor select a highest priority job from the ready queue before processors

[9]For now, assume that each job executes for its WCET. Handling premature job completions is addressed below.

[10]If the ready queue is empty, then the processor will idle during the next quantum.

whose jobs *may* continue execution is appropriate. In a similar vein, since the running queue is ordered by non-decreasing priority and has one entry for each processor, if a processor's current (not dummy) job reaches the running queue's head and has lower priority than that currently at the head of the ready queue, then this job is guaranteed to be *not* scheduled in the next quantum, and so, deleting it from the running queue (after enqueuing it in the ready queue) is appropriate. Finally, running queues of at most two consecutive quanta will be needed at any time, and so, maintaining just two queues and switching their roles at every quantum suffices. (In other words, queues for the different quanta can be maintained as a circular queue of queues of length two.)

The above approach can be made to be robust in the face of jobs not executing for their full WCET as follows. To account for the case wherein jobs complete early *exactly* at quantum boundaries, a pre-scheduling phase or *pre-phase*, in which processors of such jobs make their selections before the remaining processors, can be included. Processors selecting during the pre-phase will not select during the regular phase; the regular phase can commence after processors have reached consensus (*e.g.*, through barrier synchronization) that they are through with the pre-phase. On the other hand, if a job's early completion is within a quantum (as opposed to a quantum boundary), then when the scheduler is invoked at the early completion time[11] and a new job is selected, the processor can modify its entry in the ready queue to correspond to the newly-scheduled job or a dummy job (by changing the entry's priority and other details such as task and job identifiers) in $\mathcal{O}(\lg M)$ time (assuming a binomial or binary heap implementation). One important caveat of the above approach, however, is that if quanta are not aligned on processors, then to ensure both minimal processor waiting times and scheduling contiguous job executions on the same processor, scheduling may have to be centralized, with the first processor arriving at the beginning of a logical quantum serving as the central processor. Hence, additional synchronization and other mechanisms will be needed to communicate early job completions, wherein jobs execute for less than their WCETs, to the central scheduler (to account for scenarios in which the central scheduler makes selections assuming that a job will continue, whereas the job completes early *after* the scheduler's decision).

Though the complexities described above will have to be addressed only at those quanta at which new jobs that have higher priority than one or more of the executing jobs are released, and hence, the number of "complex" quantum boundaries is bounded by the number of jobs,

[11]The amount of time elapsed since the beginning of that quantum can be determined with high precision using high-resolution timing devices such as the Pentium® Time-Stamp Counter of Intel®.

it is not straightforward, and perhaps impossible, to distribute this incurred overhead evenly among all the jobs as is done with certain other types of overhead. Hence, the worst case is to assume that the extra overhead is incurred by every job at every quantum boundary, and charge each job based on the number of quanta it spans. Of course, the exact magnitude of the overhead can be lowered through efficient implementations.

Algorithm EDF-sliced. Recall from Chapter 8 that under EDF-sliced, each job of a task is sliced into equal-sized sub-jobs (with the possible exception that the execution cost of the final sub-job is different), which are then scheduled under g-NP-EDF. Thus, overhead accounting under EDF-sliced is similar to that under g-NP-EDF with the following differences. Let $T.s$ denote the number of sub-jobs of T. Then, since each job of T is scheduled $T.s$ times, each job needs to be charged with $T.s$ scheduling and context-switching overheads. Further, each job can suffer from cache-related preemption or migration overhead whenever a second or a later sub-job begins execution. Hence, WCET for T under EDF-sliced is given by the following. (Overhead due to tick-scheduling should be accounted for as described earlier after the inflation due to the remaining overheads is computed. Sub-job execution costs should be determined after the tick-scheduling overhead is also accounted for even though the "number" of sub-jobs, $T.s$, is determined before accounting for any overhead.)

$$T.e = T.e^{(b)} + T.s \cdot (\mathsf{oh}_{\mathsf{sch}}(\mathsf{EDF\text{-}sliced}) + \mathsf{oh}_{\mathsf{cs}}(\mathsf{EDF\text{-}sliced})) + (T.s - 1) \cdot \mathsf{oh}_{\mathsf{mig}}(\mathsf{EDF\text{-}sliced}, T)$$

Algorithm p-EDF. Accounting under p-EDF differs from accounting under g-EDF in the following. First, no job incurs cache-related migration overhead, but only cache-related preemption overhead. Hence, while charging each job J_1 with the overhead associated in resuming a lower-priority job J_2, apart from charging J_1 with one scheduling and context-switching cost, it is sufficient to charge one cache-related preemption cost (as opposed to a migration cost). Next, assuming that no job is early released, as argued in [111], it can be shown that if job J_2 of task U resumes immediately after job J_1 of task T completes, then $U.D > T.D$ holds. Therefore, only tasks with relative deadlines larger than that of T need to be considered in determining the amount of cache-related overhead to charge. Finally, since tasks do not migrate, it may appear that it is sufficient to consider only tasks assigned to the same processor as that of J_1 in determining this overhead. However, note that inflated WCETs of tasks are needed in determining a feasible partition, and accounting for overheads after determining a partitioning may render it infeasible. Therefore, if only tasks assigned to a processor are to be considered in determining the preemption overhead to charge, then, since part of the accounting has to be postponed until after a partition is determined, re-partitioning may be needed. Hence, for simplicity, we consider all the tasks in determining how much cache-related preemption cost to charge. The cache-related preemption cost to charge to task T is thus given by $\max_{\{U \mid U.D>T.D\}} \mathsf{oh}_{\mathsf{pr}}(\mathsf{p\text{-}EDF}, U)$, and the inflated WCET of T by

$$T.e = T.e^{(b)} + 2 \cdot \mathsf{oh}_{\mathsf{sch}}(\mathsf{p\text{-}EDF}) + 2 \cdot \mathsf{oh}_{\mathsf{cs}}(\mathsf{p\text{-}EDF}) + \max_{\{U \mid U.D>T.D\}}(\mathsf{oh}_{\mathsf{pr}}(\mathsf{p\text{-}EDF}, U)). \quad (9.1)$$

Under p-EDF, the worst-case tick-scheduling cost is given by the time taken to merge a per-processor release or pending queue.

Algorithm EDF-fm. For both fixed and migrating tasks, inflation needed under EDF-fm differs from that needed for any task under p-EDF in the final term of (9.1) given above. This is because a migrating task has a higher priority than any fixed task, and hence, a preempted job of a migrating task would not wait for a job of a fixed task to complete execution and resume after it. Hence, in charging a fixed task with the overhead incurred in resuming a lower priority job, migrating tasks need not be considered. For a migrating task T, the difference is due to the following. Since any fixed task, regardless of its relative deadline, assigned to either of T's processors has a lower priority than T, all these tasks need to be considered in determining the preemption overhead to charge to T. As before, since inflated execution costs and utilizations are needed in determining task assignments, it is simpler to consider all the tasks while charging overhead costs related to resuming lower-priority jobs. Hence, inflation under EDF-fm is given by the following.

$$T.e = \begin{cases} T.e^{(b)} + 2 \cdot \mathsf{oh}_{\mathsf{sch}}(\mathsf{EDF\text{-}fm}) + 2 \cdot \mathsf{oh}_{\mathsf{cs}}(\mathsf{EDF\text{-}fm}) + \\ \max_{\{U \mid U \text{ is fixed and } U.D>T.D\}}(\mathsf{oh}_{\mathsf{pr}}(\mathsf{EDF\text{-}fm}, U)) & , \ T \text{ is fixed} \\ T.e^{(b)} + 2 \cdot \mathsf{oh}_{\mathsf{sch}}(\mathsf{EDF\text{-}fm}) + 2 \cdot \mathsf{oh}_{\mathsf{cs}}(\mathsf{EDF\text{-}fm}) + \\ \max_{\{U \in \tau\}}(\mathsf{oh}_{\mathsf{pr}}(\mathsf{EDF\text{-}fm}, U)) & , \ T \text{ is migrating} \end{cases}$$

Tick-scheduling overhead under EDF-fm is the same as that under p-EDF.

Algorithm EPDF. Accounting under EPDF is similar to accounting under PD^2 presented in [111]. Like PD^2, EPDF schedules quantum-length subtasks, and if some subtask completes execution in the middle of a quantum, idles the associated processor for the remainder of the quantum. Hence, if the WCET of a task is not an integral number of quanta, then some quanta may be partially wasted, and thus, in comparison to the algorithms considered above, EPDF can incur an additional overhead. This overhead can be accounted for by rounding each task execution cost (obtained after accounting for the remaining overheads) up to the next integer, and hence is referred to as *rounding overhead*.

Rounding overhead also necessitates the use of an iterative formula for determining the overall inflation needed for each task. To see this, let $T.e^{(0)} \stackrel{\text{def}}{=} T.e^{(b)}$ denote the base WCET of T. Then, the number of scheduling decisions needed for each job of T is $\lceil T.e^{(0)} \rceil$. Similarly, since each job may be preempted at the end of every quantum, a safe upper bound on the number of preemptions is $\lceil T.e^{(0)} \rceil - 1$. Hence, the number of context switches and migrations (because each preemption may result in a migration) to account for are $\lceil T.e^{(0)} \rceil$ and $\lceil T.e^{(0)} \rceil - 1$, respectively. Accounting for these, we arrive at $T.e = T.e^{(0)} + \lceil T.e^{(0)} \rceil \cdot (\mathsf{oh}_{\mathsf{sch}}(\mathsf{EPDF}) + \mathsf{oh}_{\mathsf{cs}}(\mathsf{EPDF})) + (\lceil T.e^{(0)} \rceil - 1) \cdot \mathsf{oh}_{\mathsf{mig}}(\mathsf{EPDF}, T)$, and accounting for the rounding overhead, we have $T.e = \lceil T.e \rceil$. Note that if $\lceil T.e \rceil$ is different from $\lceil T.e^{(0)} \rceil$ then our inflation may not be safe and has to be revised by substituting $T.e$ for $T.e^{(0)}$ within the ceiling operators to arrive at a new estimate for $T.e$. The procedure has to be repeated iteratively either until convergence is reached or $T.e = T.p$, whichever is earlier. An iterative formula for this is given below. In this formula, $T.e^{(k)}$ denotes the WCET after the k^{th} iteration. As mentioned earlier, $T.e^{(0)} = T.e^{(b)}$. (It is assumed that the units for the overhead costs are quanta and can be non-integral.)

$$T.e^{(k+1)} = \left\lceil T.e^{(b)} + \lceil T.e^{(k)} \rceil \cdot (\mathsf{oh}_{\mathsf{sch}}(\mathsf{EPDF}) + \mathsf{oh}_{\mathsf{cs}}(\mathsf{EPDF})) + (\lceil T.e^{(k)} \rceil - 1) \cdot \mathsf{oh}_{\mathsf{mig}}(\mathsf{EPDF}, T) \right\rceil$$

Expressed another way, T's inflated WCET $T.e$ under EPDF is given by

$$\begin{aligned} T.e \;\; &= \;\; \min\Big(T.p, \\ &\Big(\min t : t = \Big\lceil T.e^{(b)} + \lceil t \rceil \cdot (\mathsf{oh}_{\mathsf{sch}}(\mathsf{EPDF}) + \mathsf{oh}_{\mathsf{cs}}(\mathsf{EPDF})) + (\lceil t \rceil - 1) \cdot \mathsf{oh}_{\mathsf{mig}}(\mathsf{EPDF}, T) \Big\rceil \Big)\Big) \end{aligned}$$

A difference in the above formula with respect to that provided in [111] for PD^2 is in the last term. Under PD^2, since no deadlines are missed, a better estimate on the number of preemptions is possible by noting that each job of T completes execution within $T.p$ quanta of its release. Hence, if $T.p < 2 \cdot T.e$, then each job of T can be preempted at most $T.p - T.e$ times, which is at most $T.e - 1$. Thus, the number of preemptions under PD^2 is bounded by $\min(T.e - 1, T.p - T.e)$. However, since deadlines can be missed under EPDF, each job of T may not complete until before $T.p+q$ quanta after its release, where q is a tardiness bound for the task system under consideration. Therefore, only a weaker bound of $\min(T.e-1, T.p+q-T.e)$ on the number of preemptions can be shown to hold under EPDF.[12] However, since a bound on q requires a bound on the maximum task weight, which in turn depends on the overheads, for simplicity, we just use the looser bound of $T.e-1$.

9.4 Performance Evaluation

In this section, we present the results of experiments conducted to evaluate the algorithms considered in this dissertation and p-EDF. In these experiments, base WCETs and periods for tasks were generated randomly. On the other hand, overhead costs were assigned based on measurements obtained from an experimental test-bed containing prototype implementations of the algorithms. As mentioned earlier, our objective is to evaluate the algorithms with respect to the percentage of task sets for which bounded tardiness can be guaranteed and the range for the guaranteed bounds. We first describe the procedure used to select overhead costs. The procedure used to generate task sets and the results of the evaluation are presented afterwards.

9.4.1 Estimation of Overheads

Worst-case values for the overheads described in Section 9.2 were measured using an experimental prototype called LITMUS$^{\text{RT}}$. LITMUS$^{\text{RT}}$ stands for **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems and its current version was developed in collaboration with Calandrino *et al.* [41], with facilitating comparison of real-time scheduling algorithms on a multiprocessor as one of its purposes.

LITMUS$^{\text{RT}}$ was not built from scratch, but rather by modifying Linux such that its underlying scheduler is configurable at boot time. This was facilitated by modularizing the Linux scheduler so that a new algorithm can be plugged in easily. Currently, implementations are available for g-EDF, g-NP-EDF, p-EDF, PD^2, and two variants of PD^2. Further details on LITMUS$^{\text{RT}}$ and its implementation can be found in [41]. Some relevant aspects are as follows: to ensure that all sources of unpredictability are eliminated, paging is disabled; further, all pages are locked in memory, and all real-time tasks are restricted to sharing a common address space. LITMUS$^{\text{RT}}$ currently runs on the 2.6.9 version of the Linux kernel. Our measurements

[12]It should be pointed out that the bound derived for PD^2 can be shown to hold under EPDF asymptotically (that is, over long periods) for periodic task systems.

were performed on an SMP consisting of four 32-bit Intel® Xeon® processors running at 2.7 GHz with 2GB memory. Each processor had 8K instruction and data caches, and a unified 512K L2 cache. Cache lines were 64 bytes wide.

Cache-related preemption and migration costs. Recall that a worst-case scenario for cache-related delays due to preemptions and migrations occurs when each processor has some preempted task resuming on it, and each is refetching data and/or instructions evicted from the cache. Also note that the actual delay depends on the code and working-set sizes (WSSs) of the tasks. Hence, we determined worst-case preemption and migration costs for five WSSs of 4K, 32K, 64K, 128K, and 256K (in an attempt to estimate these costs for systems with varying data dependencies). (While the WSSs considered may appear to be small, these represent the amount of data that a task fetches at the beginning of a quantum, typically within one ms, concurrently with every other processor, and it is not possible to read a much larger amount of data in that time. On the other hand, if a preemption may cause much larger working sets to be lost from the cache and much higher contention, then preemptive algorithms may not be appropriate for the underlying application.)

Worst-case costs were determined by emulating worst-case scenarios. To emulate worst-case preemptions, we let each processor write a block of memory of a specified size at the same time. The time taken for the write to complete, less the time taken to write the same amount of data in the same scenario, but with all written data being locally cached, was taken as an estimate of the worst-case preemption cost. Migration costs were measured similarly, but by including the emulation of migration of data from one cache to another. This was done by reading data into the cache of one processor, and then writing that same data on a second processor. This had the effect of bringing the data into the cache of the second processor, while invalidating it in the cache of the first processor. The amount of time taken for the second processor to write its working set (less the time taken to write the same amount of data resident in cache as was done with preemption costs) was taken as an upper bound on the worst-case migration cost. Preemption costs for the five WSSs considered were found to be $16\mu s$, $67\mu s$, $115\mu s$, $230\mu s$, and $439\mu s$, while migration costs were $16\mu s$, $68\mu s$, $126\mu s$, $272\mu s$, and $561\mu s$.

Since our measurements were on a four-processor system, in our simulation studies, we used the above costs for experiments on four processors. We expect contention on the shared bus to increase at most linearly with the number of processors, so for an arbitrary number

M of processors, where $M \in \{2, 4, 8, 16\}$, we scaled the above costs linearly. Specifically, if x denoted the cost on four processors, then the cost on M processors was taken as $Mx/4$. (The discrepancies, if any, in costs thus computed for $M \neq 4$ from the actual costs that can be observed on an M processor system remain to be determined.)

Context-switching overhead. Context-switching overhead was measured by recording the times before and after a context-switch call assuming that all processors perform a context switch concurrently. This cost was found to be of the order of $0.5\mu s$ for p-EDF, and $1\mu s$ for the remaining algorithms. For all the algorithms, in comparison to the other costs, the cost of a context switch was thus negligible.

Scheduling overhead. In assigning scheduling costs, we *assumed all tasks to be periodic*. In the current version of LITMUS$^{\mathrm{RT}}$, priority queues are implemented using linked lists (which are not the most efficient way for implementing priority queues). Further, the implementation of g-EDF is not efficient. We considered these to be limiting, and hence, did not use LITMUS$^{\mathrm{RT}}$ to measure scheduling costs. Rather, we approximated scheduling costs by measuring times in a binomial heap implementation. The predominant cost in making a scheduling decision consists of switching to kernel mode, enqueuing the completing job's next instance or the preempted job in a release queue or the ready queue, merging the ready queue and a release (or the pending queue), and dequeuing the highest-priority task from the ready queue. Hence, we measured the total time taken to perform these operations in our binomial heap implementation. The heap node contained all the fields that would be present in a priority-queue node of a scheduler. A cold cache was ensured before each fresh set of operations (enqueue, merge, and extract) was performed.

We measured the average time taken to perform the entire set of operations described above, including the time taken to switch to kernel mode, for n ranging from ten to 2000, where n denotes the number of nodes. For each n, the average time taken was between $2.0\mu s$ and $3.0\mu s$, of which approximately $0.75\mu s$ is the time taken to switch to kernel mode. Since the worst-case time complexity of all the operations of interest to us is $\mathcal{O}(\lg n)$ when a binomial heap is used, we (somewhat arbitrarily) fixed the cost of a scheduling decision for a queue size of n on a single processor as $2.0 + \lg n \cdot 0.125\mu s = 0.75 + 1.25 + \lg n \cdot 0.125\mu s$. (The actual reasoning behind the second term is as follows. $\lg n$ increases approximately by 8 as n increases from 10 to 2000. Hence, since the measured time increased by $1000ns$ as n increased from 10 to 2000, it seemed appropriate to increase the total cost by $1000/8 = 125ns$ as $\lg n$ increases by one.) Thus, for p-EDF and EDF-fm, since the number of tasks per processor is N/M on average, a cost of $2.0 + \lg \frac{N}{M} \cdot 0.125\mu s$ was charged.

We next describe the cost assigned to global algorithms and the reasoning behind it. Global algorithms use a shared queue and are restricted to accessing it in a mutually-exclusive manner. Scheduling is either *centralized*, with a single central processor making scheduling decisions for all the processors as with Pfair algorithms, or *distributed*, with each processor making its own scheduling decision. Either way, in the worst case, each processor is delayed by the time taken to make $M - 1$ scheduling decisions, and hence, for global algorithms, we charged a scheduling cost of $0.75 + M \cdot (1.25 + \lg n \cdot 0.125)\mu s$ per scheduling decision. Compare this expression with the one provided above for partitioned algorithms. The $0.75\mu s$ term in this expression accounts for the time taken to switch to kernel mode. Since each processor can switch to kernel mode independently, and is not delayed by other processors, this cost need not be multiplied by M. We ignored queue-based spin lock acquisition times, as these were measured to be minimal (of the order of nanoseconds).[13] Also, since measurements were made with a cold cache, the migration of priority-queue nodes from one cache to another is accounted for. Further, measurements with scheduler implementations on LITMUS$^{\mathrm{RT}}$ show that in comparison to p-EDF, the cost of a scheduling decision is higher under global algorithms by less than 1 μs (except under g-EDF, whose LITMUS$^{\mathrm{RT}}$ implementation is not efficient). Hence, we believe that our accounting for global algorithms is safe by a wide margin.

Overhead due to tick scheduling. For all non-Pfair algorithms considered, except g-EDF, the predominant cost incurred due to tick scheduling is the time taken to switch to kernel mode, and merge a release queue or the pending queue with the ready queue. This cost was measured to be almost the same as the scheduling cost described above, and hence, we charged $2.0 + \lg n \cdot 0.125\mu s$ for g-NP-EDF and EDF-sliced, and $2.0 + \lg \frac{N}{M} \cdot 0.125\mu s$ for p-EDF and EDF-fm. For global algorithms, since at most one processor performs a merge, and lock acquisition times within the kernel (after switching to kernel mode) are negligible, per-processor delays are negligible. Hence, there is no term accounting for per-processor delays. Under g-EDF, to

[13]Typically, synchronization within the kernel is by disabling interrupts, or by "busy waiting," also known as *spinning*. Disabling interrupts is not effective (*i.e.*, cannot guarantee mutual exclusion) on multiprocessors and simple spinning cannot guarantee predictable waiting times and can cause excessive traffic on the interconnect. Queue-based spin locks [89] lower interconnect traffic by having each process spin on a variable that is local to it, and hence, each spin variable can be cached. Predictability is ensured by including mechanisms to order spinning processes in a FIFO manner and by enabling their spin variables in that order.

ensure that jobs whose executions span contiguous quanta are not migrated, in the worst case, apart from one merge by any of the processors, each processor may have to wait for every other processor to update the status of its current job, and perform one enqueue and dequeue operation, and hence, we charged $0.75 + M \cdot (1.25 + \lg n \cdot 0.125)\mu s$ as the tick-scheduling cost for g-EDF. (As with the scheduling overhead, the term $0.75\mu s$ corresponds to the time taken to switch to kernel mode, and since each processor can switch to kernel mode independently and is not delayed by any of the remaining processors, this term is not multiplied by M.)

9.4.2 Experimental Setup

We are now ready to describe our simulation procedure. Since the processors-to-memory interconnection can be quite different and need not be a bus when the number of processors is greater than 16, we do not believe that overheads measured on a four-processor system can be extrapolated to systems with greater than 16 processors. Hence, in our simulations we limited M to be at most 16. Simulations were performed for each combination of the parameters listed below for algorithms p-EDF, g-EDF, g-NP-EDF, EDF-fm, EDF-sliced, and EPDF. (PD^2 was not included as overheads under it are almost the same as that under EPDF.) As mentioned earlier, each task set was assumed to be periodic. In our LITMUS$^{\mathrm{RT}}$ implementation, all tasks share a single address space, and hence, we assume this for our experiments. as Under EDF-sliced, jobs were sliced such that the base execution cost of each sub-job (except that of a job's last sub-job) was integral and at least two quanta. The parameters below were chosen so that the two conflicting goals of ensuring that scenarios that arise in practice are covered and of maintaining a manageable simulation process are both reasonably met. Overhead costs were based on measurements described in the previous section.

- M in $\{2, 4, 8, 16\}$;
- quantum size Q in $\{500\mu s, 1000\mu s, 5000\mu s\}$;
- base task utilization limits, $[u_{\min}, u_{\max})$, in $\{[0.1, 0.5), [0.3, 0.7), [0.5, 0.9), [0.1, 0.9)\}$;
- task period limits, $[p_{\min}, p_{\max})$, in $\{[10, 100), [100, 500)\}$;
- preemption and migration costs, denoted as a pair, in $\{(4M\mu s, 4M\mu s)$, $(29M\mu s, 32M\mu s)$, $(58M\mu s, 68M\mu s)\}$ for WSSs of 4K, 64K, and 128K respectively; (all tasks of a task set

were assigned equal preemption and migration costs);[14]

- context-switching cost fixed at $2.0\mu s$;
- scheduling cost set to $2.0 + \lg(\frac{N}{M}) \cdot 0.125\mu s$ for p-EDF and EDF-fm and to $0.75 + M \cdot (1.25 + \lg(N) \cdot 0.125)\mu s$ for the remaining algorithms;
- overhead due to tick scheduling set to $2.0 + \lg(\frac{N}{M}) \cdot 0.125\mu s$ for p-EDF and EDF-fm, to $2.0 + \lg N \cdot 0.125\mu s$ for g-NP-EDF and EDF-sliced, and to $0.75 + M \cdot (1.25 + \lg(N) \cdot 0.125)\mu s$ for g-EDF.

Quantum size was primarily varied to determine the impact on EPDF and EDF-sliced. Recall that either quantum-length subtasks, or sub-jobs (of uniform size that is at least one quantum and that can be chosen arbitrarily) are scheduled under these algorithms. Hence, scheduling and migration overheads are lowered as quantum size is increased. On the other hand, for EPDF, rounding error can increase. (Since tick-scheduling overhead is less in comparison to other overheads, we do not expect the remaining algorithms to be impacted much.)

The range for task periods was varied for a similar reason. For a given base task utilization, a larger period implies a larger base execution cost. Since under the non-Pfair algorithms considered, the number of preemptions is at most the number of jobs, and all overhead, except the tick-scheduling overhead, charged to a job is independent of the job size, the overhead cost per base unit of execution decreases with increasing execution costs.

For each combination of the parameters listed above (referred to as a *run*), a certain number of random task sets was generated. (The exact number is discussed later.) The task set generation procedure was as follows. For each new task set, $M+1$ tasks were initially generated. Each task's period and utilization were distributed uniformly in the range $[p_{\min}, p_{\max})$ and $[u_{\min}, u_{\max})$, respectively. Tasks were added one at a time to the initial task set as long as the base total utilization was less than M. Upon addition of each new task, overheads were determined, and a tardiness bound was computed if the inflated utilization, after accounting for the overheads, was less than M. (The addition of tasks terminated when the inflated utilization exceeded M.) Thus, schedulability was determined for several subsets of the entire task set. The total number of complete (or independent) task sets generated depended on the per-task utilization range, and was limited to 30,000, 45,000, 60,000, and 45,000, in that order, for the four ranges listed earlier. These numbers were chosen after some trial-and-error so

[14]There was no noticeable difference in the results if these costs differed slightly for different tasks.

that enough task sets were generated for each base total utilization. The number of generated task sets had to be varied with the per-task utilizations to ensure that a sufficient number of task subsets with lower total utilizations were generated even when per-task utilizations were high. (Note that when the utilization of each task is at least 0.5, with $M + 1$ tasks, the base total utilization is at least $(M + 1)/2$.) In computing inflated WCETs, the formulas provided in Section 9.2 were used. The total number of task sets (including subsets) that each algorithm could schedule[15] was determined for each run and is plotted against varying total base utilizations. Under p-EDF, the first-fit decreasing heuristic (in which tasks are considered in non-increasing order of their utilizations) was used for partitioning tasks among processors, and under EDF-fm, the LEF (resp., LUF) task assignment heuristic was used when $u_{\max}$ (after inflation) was at most 0.5 (resp., exceeded 0.5). (Recall from Chapter 5 that LEF guaranteed the lowest tardiness if an assignment under it was possible and LUF could assign the maximum number of task sets with $u_{\max} > 0.5$.) The average of the tardiness bound guaranteed was also computed for each algorithm, and is also plotted against varying base utilizations. In computing the average tardiness bounds, only schedulable task sets were included. Hence, the tardiness results should be used only in conjunction with the schedulability results.

9.4.3 Experimental Results

Graphical plots of the results of the simulation runs described above are presented in the following manner.[16] As mentioned above, two graphs were generated for each run. The x axis represents the total base task system utilization in both the graphs; the y axis represents the number of task sets in the first graph, and the mean of maximum tardiness, normalized with respect to the average execution cost, in the second graph. The first graph plots the total number of task sets generated and the number found to be schedulable under each algorithm, while the second graph plots the mean normalized tardiness for each algorithm, considering only schedulable task sets. For example, if Algorithm $\mathcal{A}$ could guarantee bounded tardiness to 1000 task sets in some run, whereas Algorithm $\mathcal{B}$ could do so for only 10 task sets in the same run, then the tardiness bounds plotted for $\mathcal{A}$ and $\mathcal{B}$ are averages of the tardiness bounds computed for the 1000 and 10 schedulable task sets, respectively. Therefore, a lower tardiness bound plotted for $\mathcal{B}$ does not imply that $\mathcal{B}$ is necessarily "better," and hence, as mentioned

[15] In the rest of this chapter, at the risk of abusing terminology, we refer to a task set that can be guaranteed bounded tardiness under Algorithm $\mathcal{A}$ as *schedulable* under $\mathcal{A}$.

[16] Plots in which $M = 16$ or $Q = 500\mu s$ have been omitted to limit the amount of data presented.

above, tardiness results should not be considered independent of schedulability results. 99% confidence intervals are shown in the second graph. (99% confidence intervals were determined for the plots in the first graph also, but have been omitted as their ranges are minimal, and their inclusion obscures the identification marks of the different curves.)

The generated graphs are grouped as follows. Four graphs, one each for each task utilization range, and equal values for the remaining parameters (such as periods, quantum size, *etc.*), are grouped and are shown in a common figure in a single page. In each figure, all four graphs (or insets (a) through (d)) plot either the number of schedulable task sets or the mean of the maximum tardiness. The task utilization range is the lowest for inset (a), highest for inset (c), and in between for inset (b). In inset (d), task utilizations span the range $[0.1, 0.9)$, which includes the ranges of the other three insets. For each set of parameters, the figure with schedulability plots is presented first followed by the figure with tardiness plots. This should facilitate comparing the tardiness bounds to the percentage of task sets for which the bound is applicable. It should be noted that since the tardiness bounds plotted are average values computed considering only schedulable task sets, a drop (which may seem counterintuitive) can be observed with increasing base utilization for some algorithms, most notably for EDF-fm (discussed further later). This drop is due to a steep decrease in the number of schedulable task sets. For the same reason, as described above, a lower tardiness bound does not imply better overall performance. Another aspect that needs explanation is the variation in the shape of the "Total Task Sets" curve, and hence, those of the other curves, with varying per-task utilizations. Note that in almost every figure with schedulability plots, the "Total Task Sets" curve is a straight line in inset (a) (*i.e.*, for low per-task utilizations), and is in general (except at the extreme end) a non-decreasing function of the base total utilization in insets (b)–(d). This variation is due to the difficulty in generating sufficient number of task sets with low base total utilizations when per-task utilizations are high, as explained earlier.

Figures are presented in increasing order of M. For each choice of M, figures are shown for the three preemption and migration costs, and the two period ranges considered. In what follows, we discuss performance trends exhibited by the algorithms as parameters are varied.

Impact of task utilizations. To study the impact of task utilizations, we will consider the results in Figures 9.6 and 9.7. For the runs here, $M = 2$, and preemption and migration costs are the lowest (of those considered). Referring to inset (a) of Figure 9.6, when task utilizations are low, schedulability is 100% for all the algorithms even when the total base utilization exceeds 75%. In this inset, schedulability drops first for EPDF, followed by p-EDF and EDF-sliced in that order. The remaining three algorithms (g-EDF, g-NP-EDF, and EDF-fm) could schedule 100% of the task sets even when the base load is 90%. However, because g-NP-EDF does not suffer from job preemptions or migrations, and EDF-fm incurs lower scheduling overhead, these two algorithms perform better than g-EDF by over 25% when the base load is 95%. The main difference between EPDF and EDF-sliced is in the absence of rounding overhead for the latter, and hence, the difference in the schedulabilities of these two algorithms is a rough indicator of the loss due to rounding. Similarly, since preemption costs are minimal here, the loss suffered by p-EDF is primarily due to an inability to feasibly partition a task set. Note that even when both task utilizations and preemption costs are low, which is the best case for p-EDF, g-NP-EDF and EDF-fm perform better than p-EDF by over 50%, and g-EDF performs better by 20%, when the total load is 1.9, which is near the maximum allowed.

As we move to inset (b), schedulability decreases for p-EDF, and in inset (c), it drops to close to 0%. However, there is no significant change for the remaining algorithms. (As mentioned earlier, the variations in the shapes of the "Total Task Sets" curve and other curves with per-task utilization ranges is due to the variation in the total number of task sets generated for each base total utilization, and do not indicate differences in schedulability.) Note that EDF-fm performs strikingly well even with high task utilizations. This is due to fact that $M = 2$ here, and on two processors, all feasible task systems can be successfully assigned (refer to Chapter 5).

Considering the tardiness bounds guaranteed by the algorithms, the three algorithms with higher schedulability suffer from higher tardiness as well. p-EDF performs well only when per-task and total utilizations are both low. However, this is not an interesting scenario. EPDF performs well (when migration costs are low to moderate) even when total utilization is moderately high regardless of task utilizations. On the other hand, if per-task or total utilization or both are high, g-EDF, g-NP-EDF, or EDF-fm may be used depending on tardiness tolerance limits. Referring to inset (a) of Figure 9.7, at low task utilizations, tardiness is lowest for EDF-fm; when task utilizations are high, g-EDF performs better. In all cases, schedulability is highest for g-NP-EDF, implying that g-NP-EDF may be the only choice for some task systems.

Similar trends are observed for all of the algorithms except EDF-fm in the other figures also, that is, when task utilizations are varied for other values (than considered above) of the remaining parameters also. (Note that we are referring only to the changes observed among the insets in one of Figures 9.19–9.43, and not to the changes between two different figures.) For

$M > 2$, schedulability drops significantly for EDF-fm when moving from inset (b) to inset (c), that is, when all tasks are heavy. For example, refer to insets (b) and (c) of Figure 9.19. Note that the performance of EDF-fm is significantly better than that of p-EDF in insets (b) and (d), even though not all tasks are light in these task sets. (Recall from Chapter 5 that bounded tardiness is guaranteed under EDF-fm only if all tasks are light.)

Two trends warrant explanation in the tardiness figures. First, tardiness plots are not smooth for g-EDF and g-NP-EDF for any M, but contain jumps. The expressions for tardiness bounds for these algorithms contain $\lfloor U_{sum} \rfloor$ as part of the upper limit of the summation in the first term of the numerator and the second term of the denominator, and hence, the tardiness bound changes abruptly at each integral value of U_{sum}. Since U_{sum} denotes the total inflated utilization and the plots are against the total base utilization, which is less than the inflated utilization, abrupt jumps can be observed at non-integral values of the total base utilization. (The next integral value would represent the corresponding total inflated utilization.) The second trend of interest is that when $M > 2$, the tardiness bound for EDF-fm is not strictly non-decreasing in insets (b) and (c), and in some cases in inset (d). Specifically, some undulations can be observed in insets (b) and (d), and a steep drop when the total base utilization is in the 80%-85% range in inset (c). In insets (b)–(d), the maximum inflated utilization of a task exceeds 0.5, and hence, the LUF heuristic is used for task assignment for task sets in these insets. We surmise that the undulations, and to some extent, the steep drop, are due to the uniform distribution of the spare capacity among the processors when assigning tasks. Hence, as the base total utilization is only slightly increased, a slightly higher percentage of each processor's capacity is made available for assignment to tasks, and so, for a given per-task utilization range, it may be possible to lower the utilizations of migrating tasks in comparison to when the total base utilization is slightly lower. Since the tardiness bound under EDF-fm increases with the utilizations of the migrating tasks, a decrease in the tardiness bound can ensue. To see this, suppose that a task system's maximum inflated per-task utilization and total inflated utilization are 0.8 and 0.75M, respectively. Then, because the distribution of spare capacity across processors is uniform, each processor is utilized only up to 75%. Hence, no task with utilization exceeding 0.75 can be assigned as a fixed task. On the other hand, if the total utilization is increased to 0.8M, then since the total utilization on each processor can be up to 80%, and under LUF, fixed tasks are assigned in the order of non-increasing utilizations, there is a significantly higher scope for tasks with utilizations in the range $(0.75, 0.8]$ to be assigned as fixed tasks. On the other hand, task assignment becomes harder as the total utilization increases, and so, whether tardiness increases or decreases depends on which of the two factors dominates as the base total utilization is slightly increased, and can be arbitrary and difficult to explain around certain points. We also surmise that the tardiness plots will be smoother if the task assignment heuristic is modified so that the uniform distribution of spare capacity guideline is relaxed to facilitate assigning heavier tasks as fixed tasks. The steep drop in the tardiness bound in inset (c) could additionally be due to the deep plunge in schedulability that accompanies at the corresponding point. Recall that the tardiness bound plotted is the average computed for schedulable task sets only.

Bimodal distribution for task utilizations. In all the schedulability figures, when *all* tasks are heavy (*i.e.*, in inset (c)), even EPDF and EDF-sliced — the algorithms that incur the most overhead — perform better than p-EDF and EDF-fm. However, since the practical significance of task systems composed exclusively of heavy tasks is not known, we also determined the schedulability for task systems that are not exclusively but only predominantly heavy, by including a few light tasks using a bimodal distribution for task utilizations, distributed between the ranges $[0.1, 0.5)$ and $[0.5, 0.9)$ with probabilities 0.1 and 0.9, respectively. Schedulability results for this bimodal distribution for task utilizations are available in Figures 9.18, 9.31, and 9.44, for $M = 2$, $M = 4$, and $M = 8$, respectively. Though schedulability is much higher for p-EDF and EDF-fm in these figures than when all tasks are heavy, EPDF and EDF-sliced are still *always* better than p-EDF, and are better than EDF-fm in most cases. Schedulability can be expected to improve for p-EDF and EDF-fm as the percentage of light tasks is increased and deteriorate for EPDF and EDF-sliced. Insets (d) of the schedulability figures, in which task utilizations are uniformly distributed in $[0.1, 0.9)$, correspond to the case when the percentages of light and heavy tasks are nearly equal.

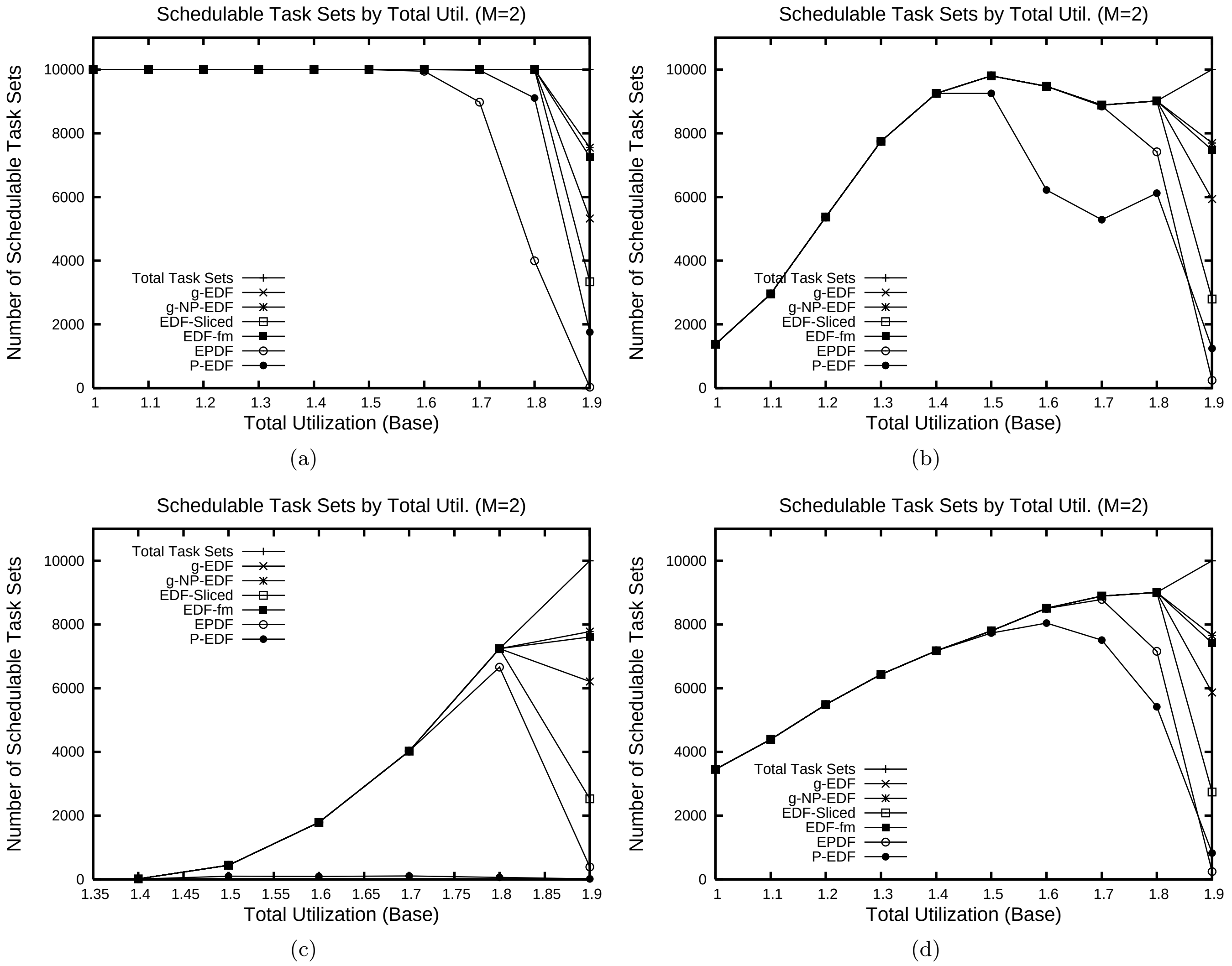


Figure 9.6: Schedulability comparison for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $8\mu s$, and migration cost = $8\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

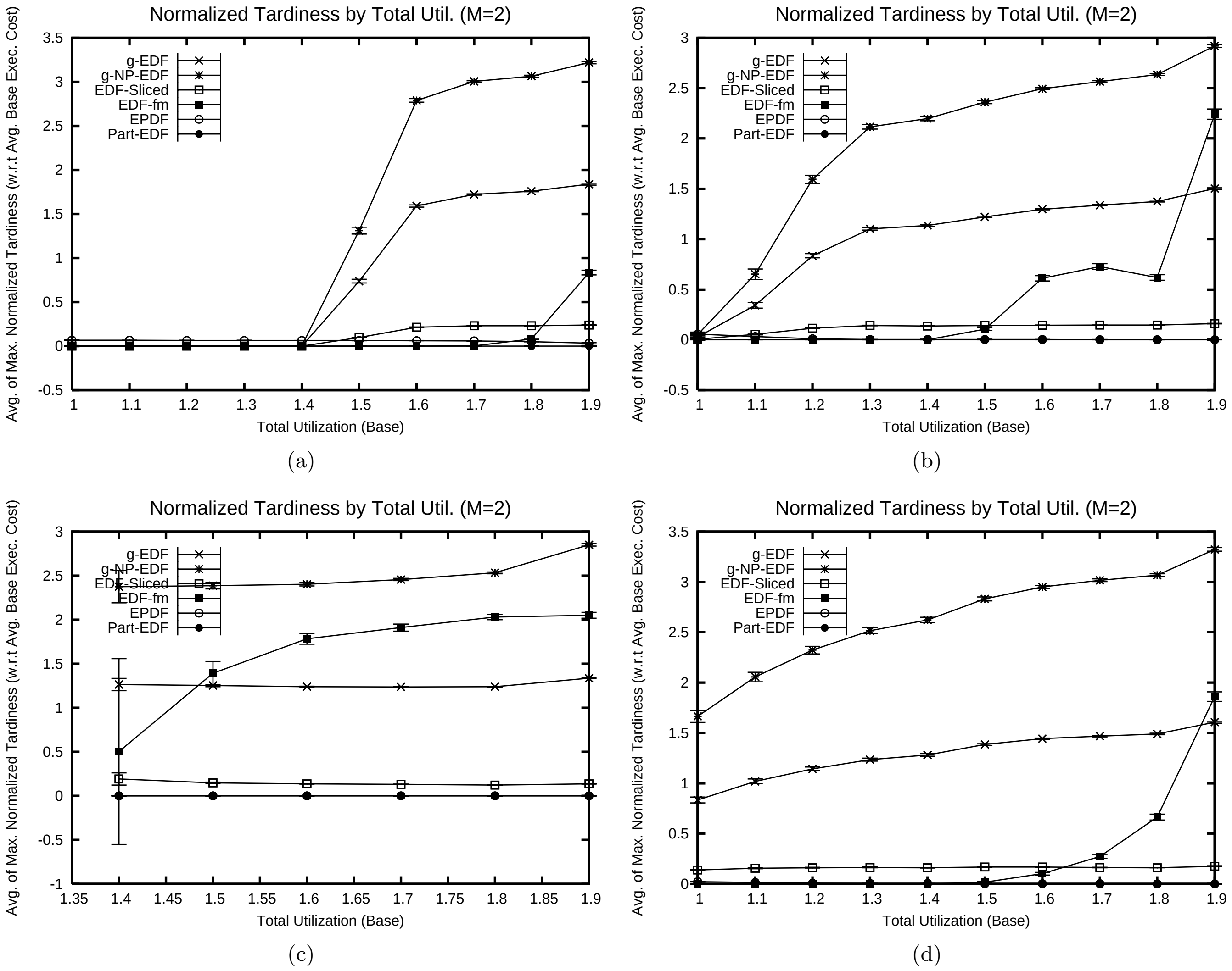


Figure 9.7: Comparison of tardiness bounds for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $8\mu s$, migration cost = $8\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

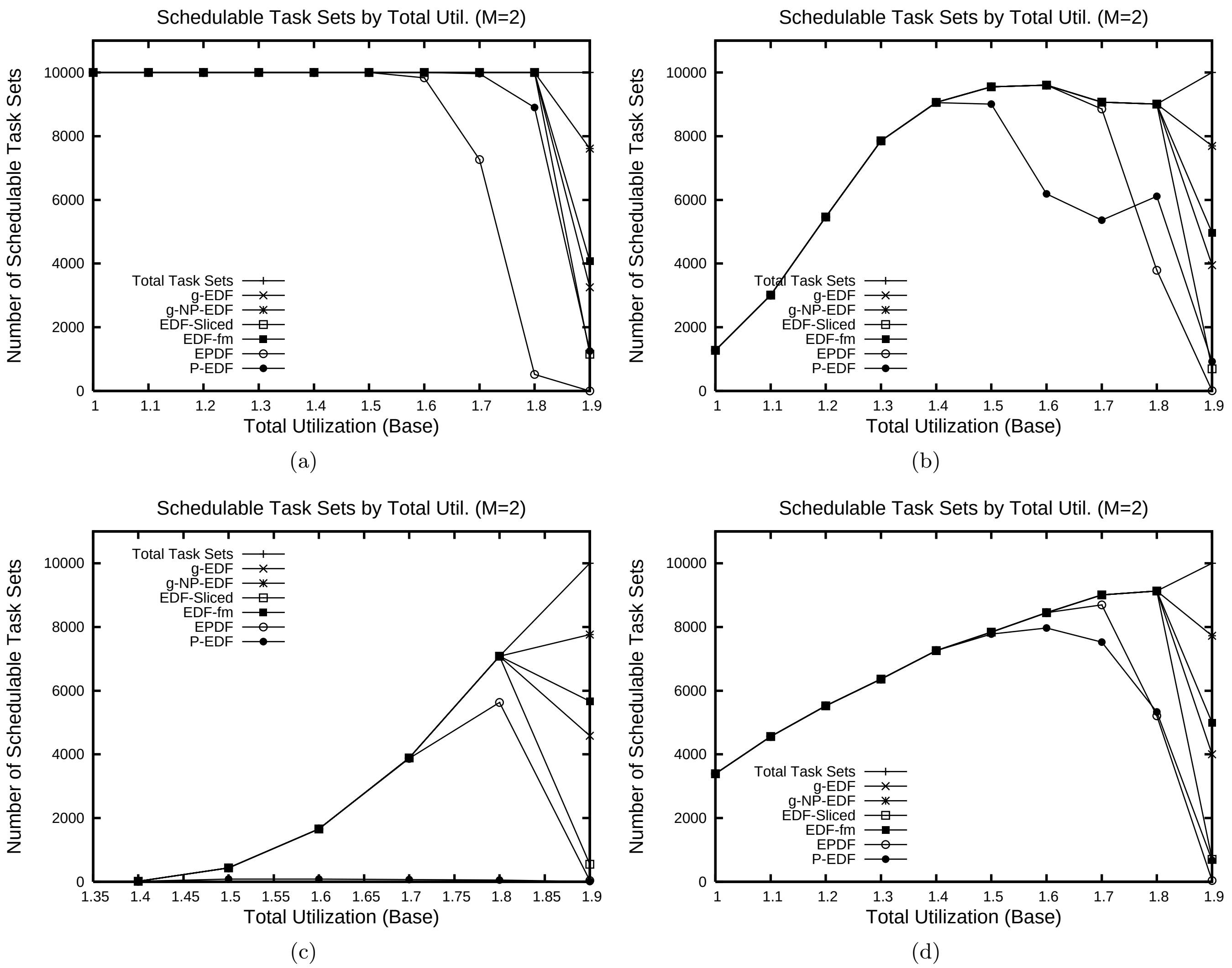
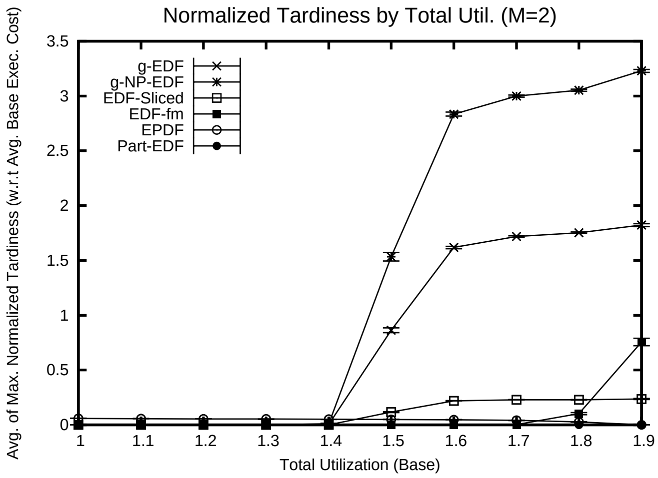
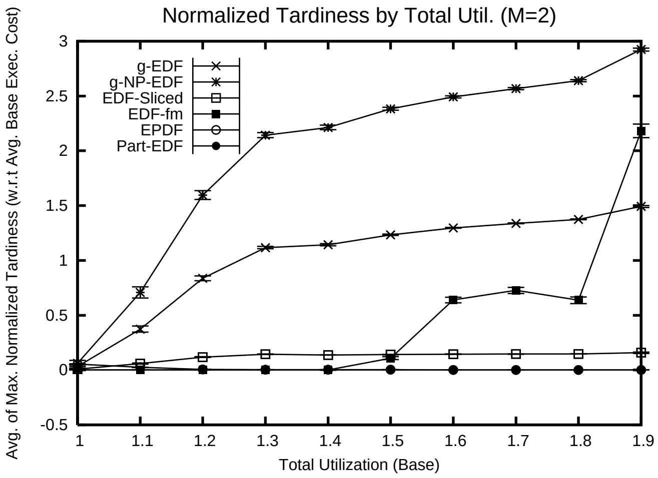


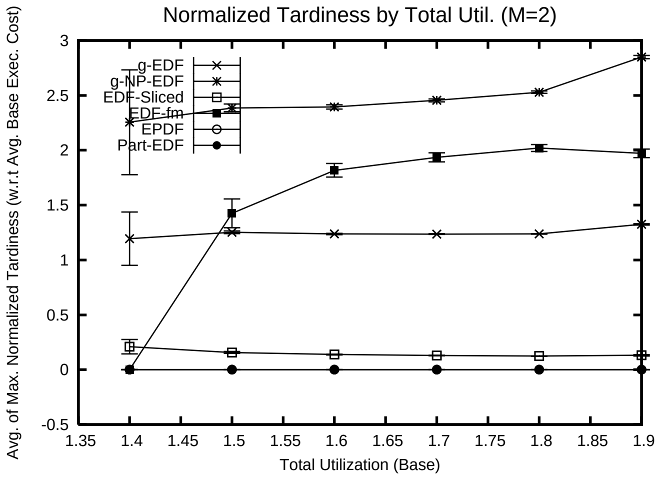
Figure 9.8: Schedulability comparison for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, $\mathrm{WSS} = 64\mathrm{K}$ (preemption cost $= 58\mu s$, migration cost $= 64\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.



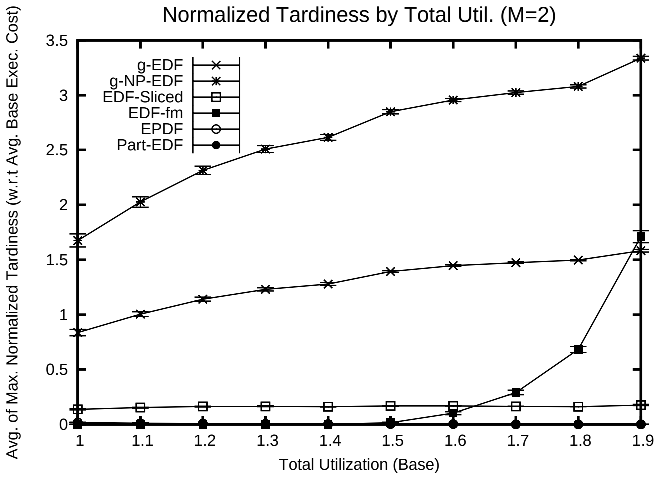
(a)



(b)



(c)



(d)

Figure 9.9: Comparison of tardiness bounds for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 64K (preemption cost = $58\mu s$, migration cost = $64\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

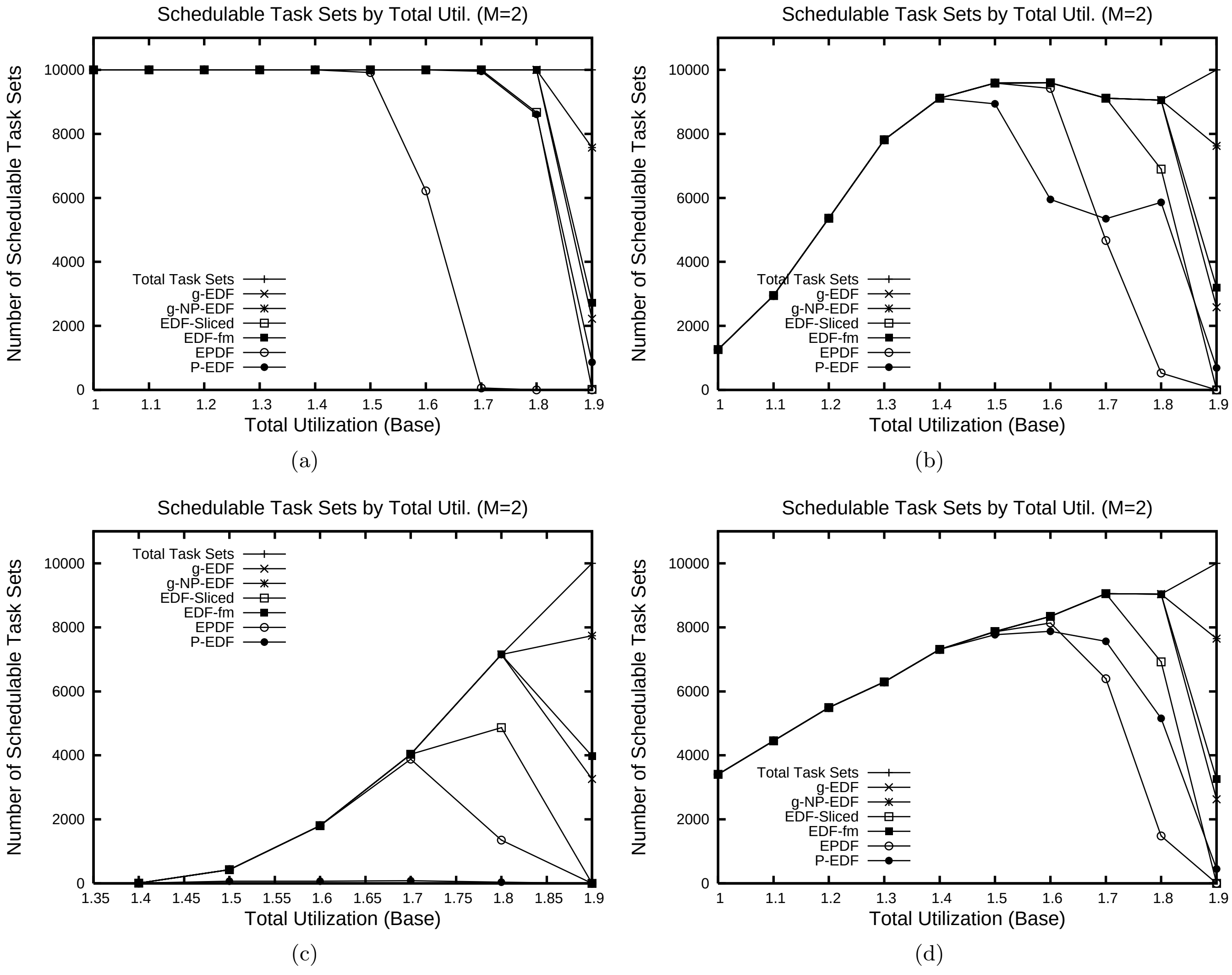


Figure 9.10: Schedulability comparison for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $116\mu s$, migration cost = $136\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

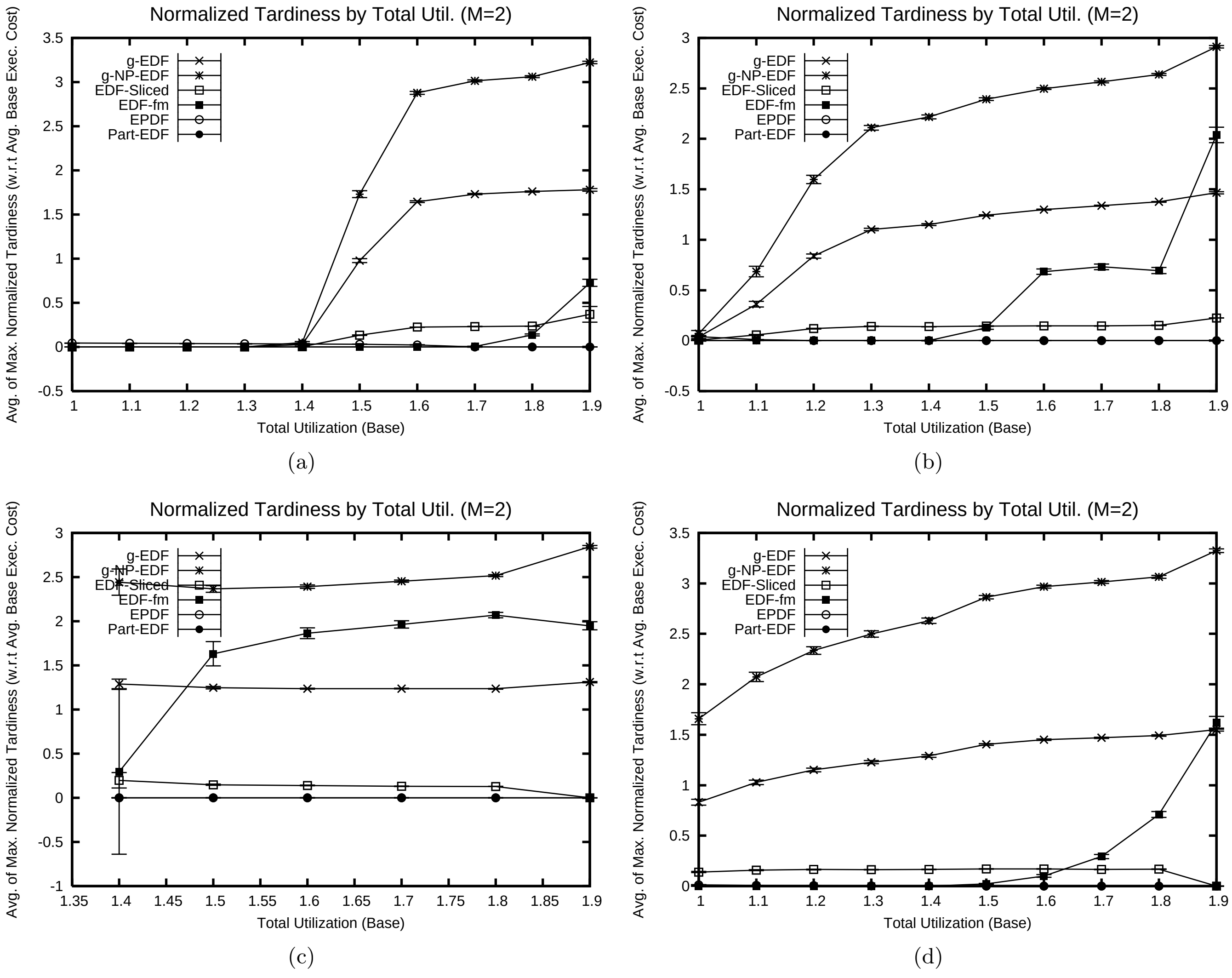


Figure 9.11: Comparison of tardiness bounds for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $116\mu s$, migration cost = $136\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

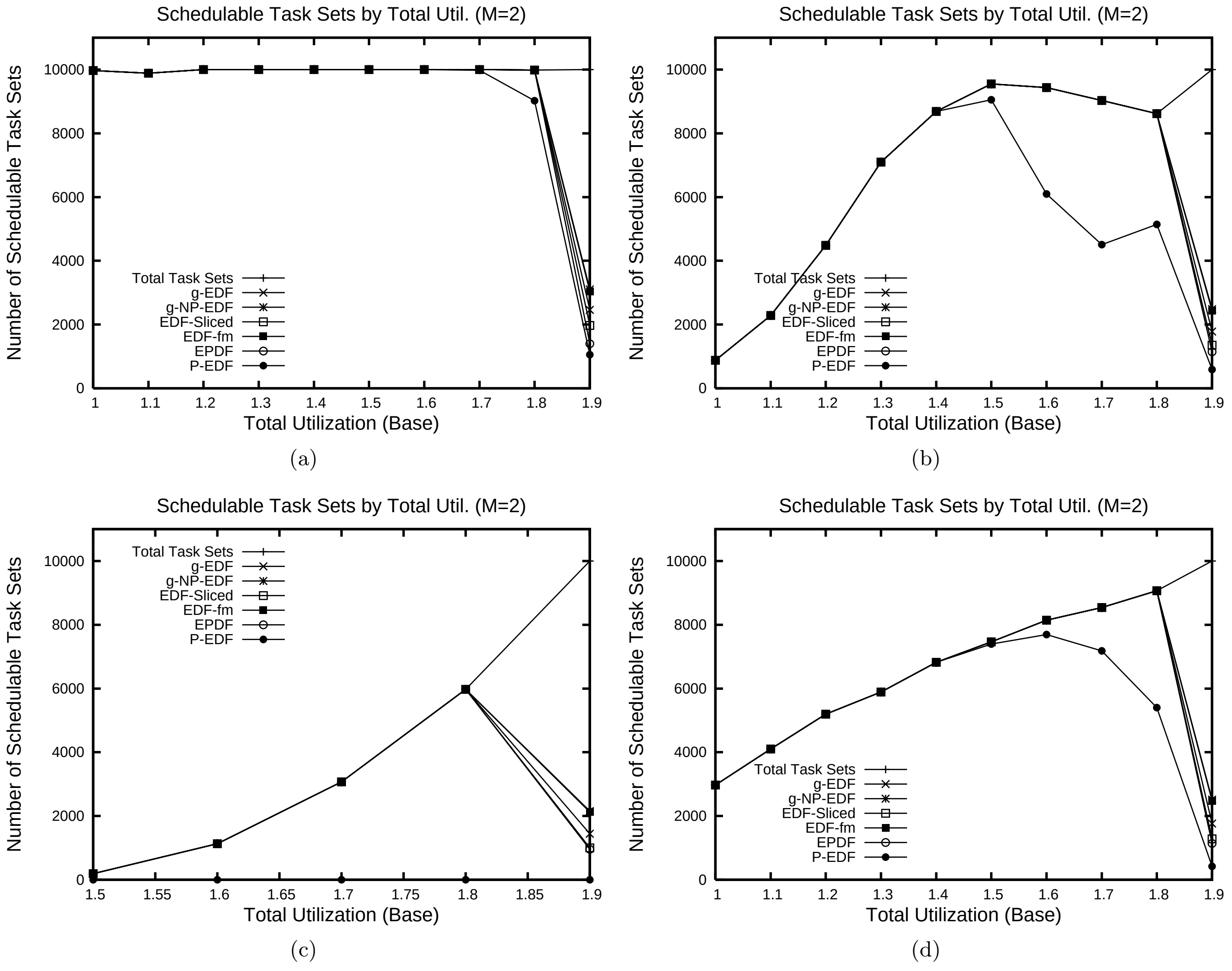


Figure 9.12: Schedulability comparison for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $8\mu s$, and migration cost = $8\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

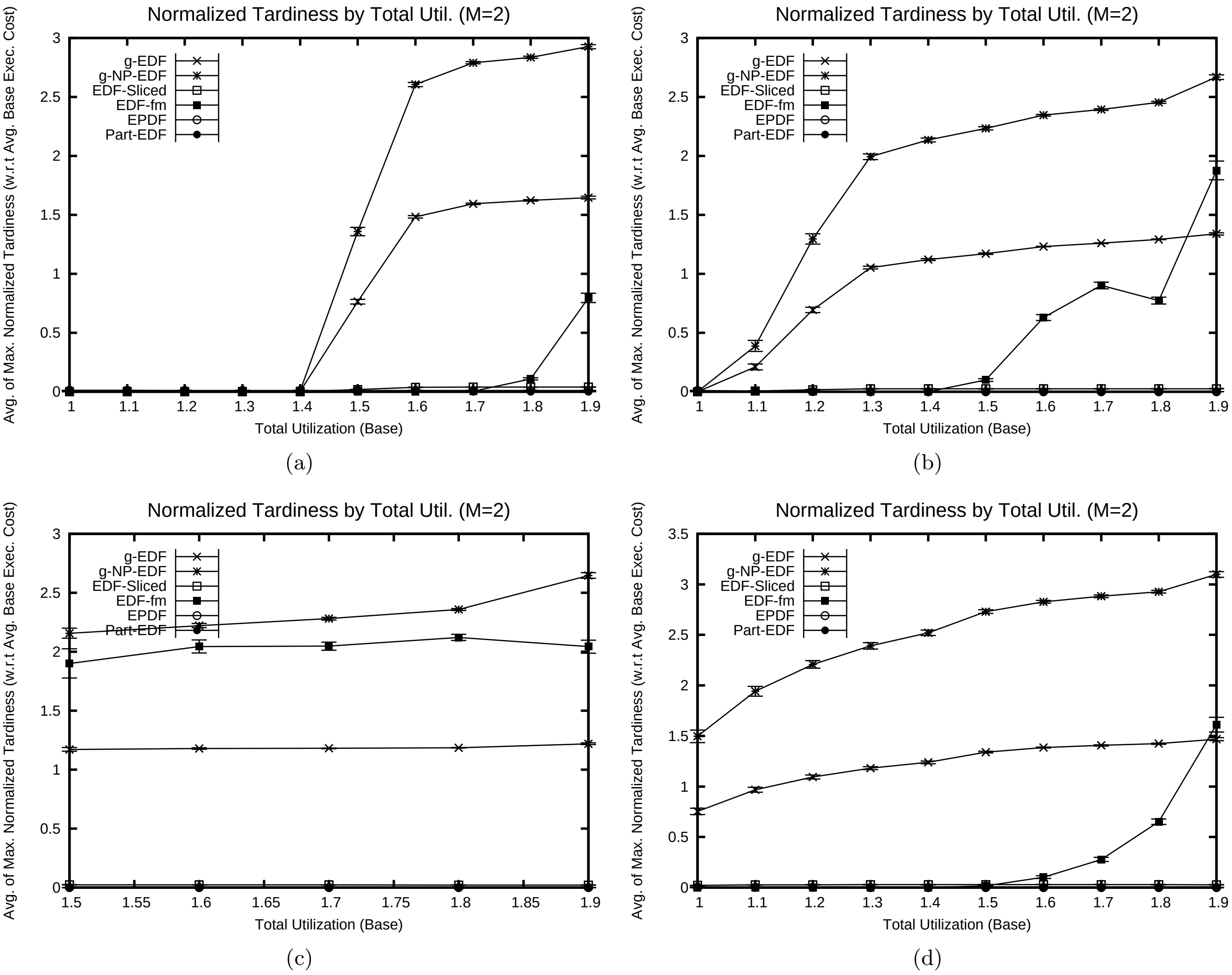
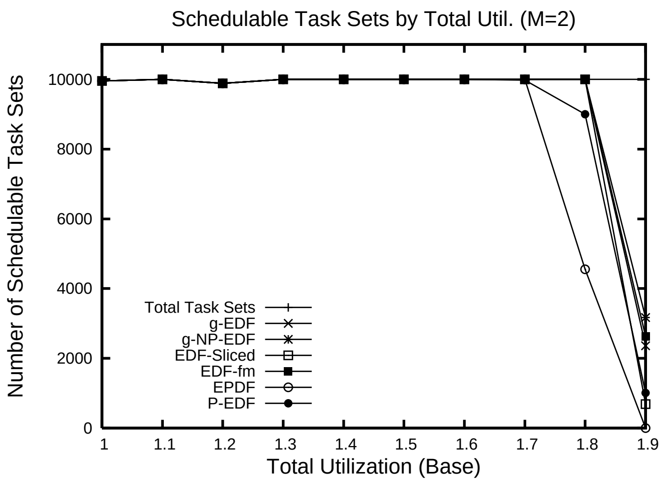
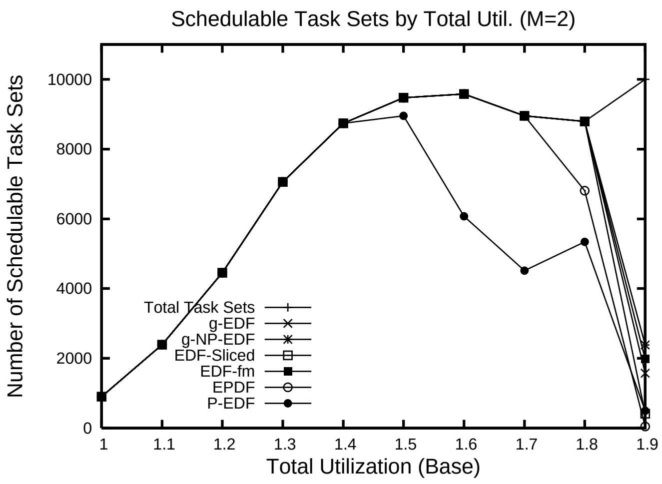


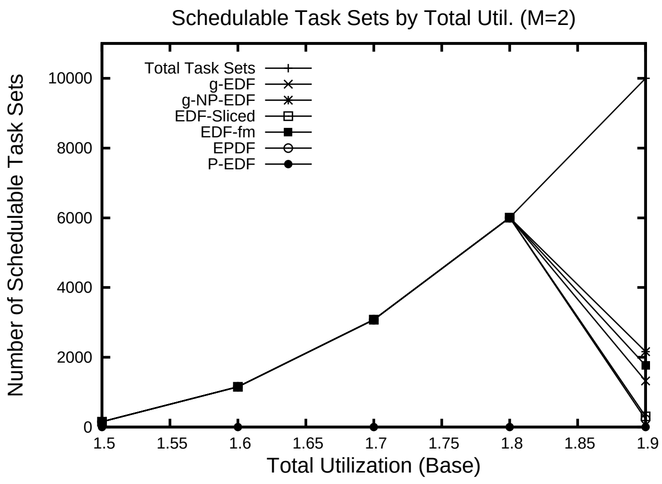
Figure 9.13: Comparison of tardiness bounds for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $8\mu s$, migration cost = $8\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.



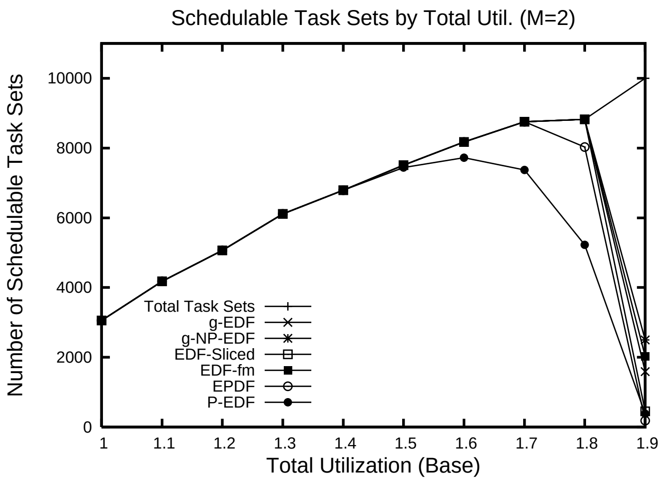
(a)



(b)



(c)



(d)

Figure 9.14: Schedulability comparison for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $58\mu s$, migration cost = $64\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

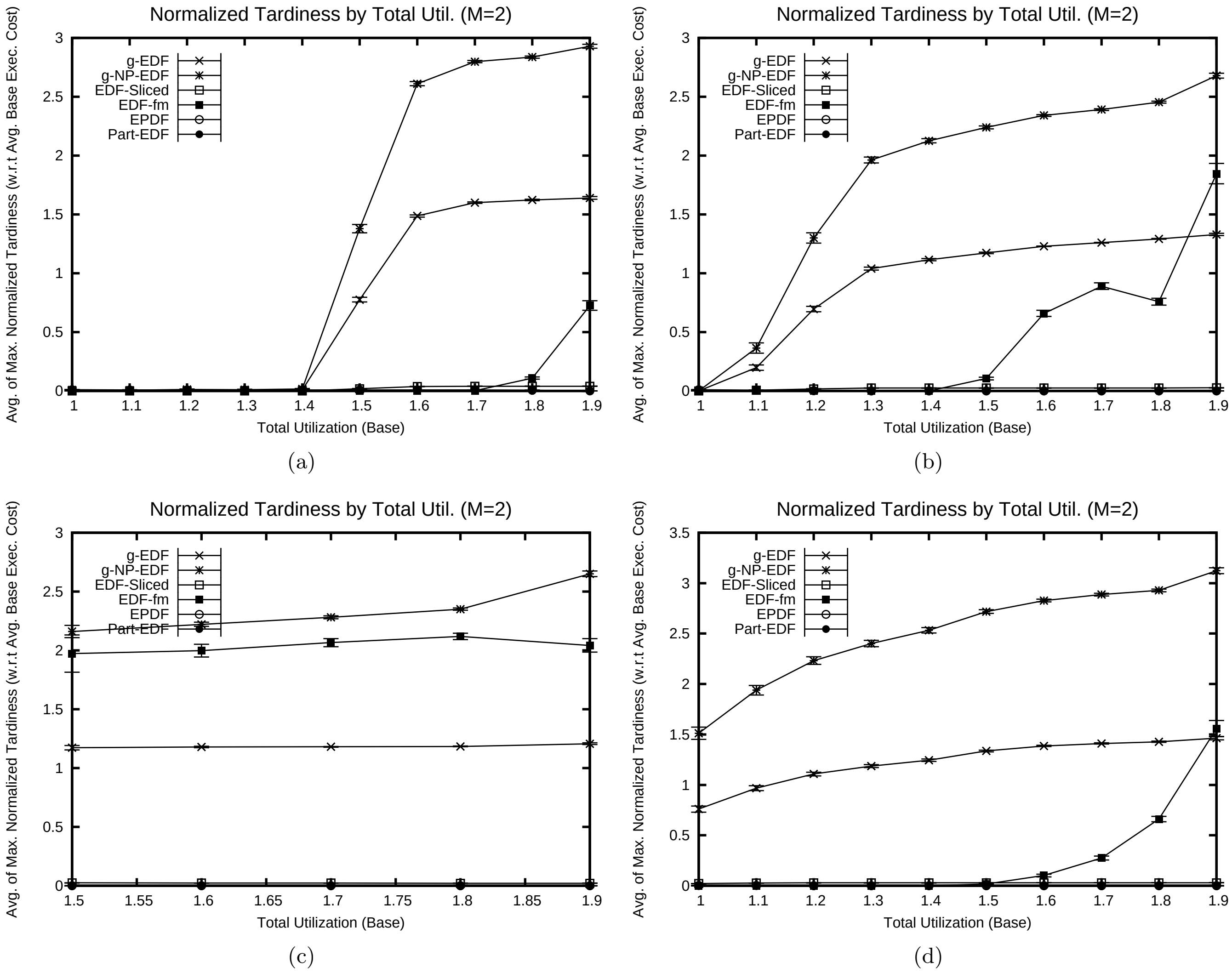


Figure 9.15: Comparison of tardiness bounds for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $58\mu s$, migration cost = $64\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

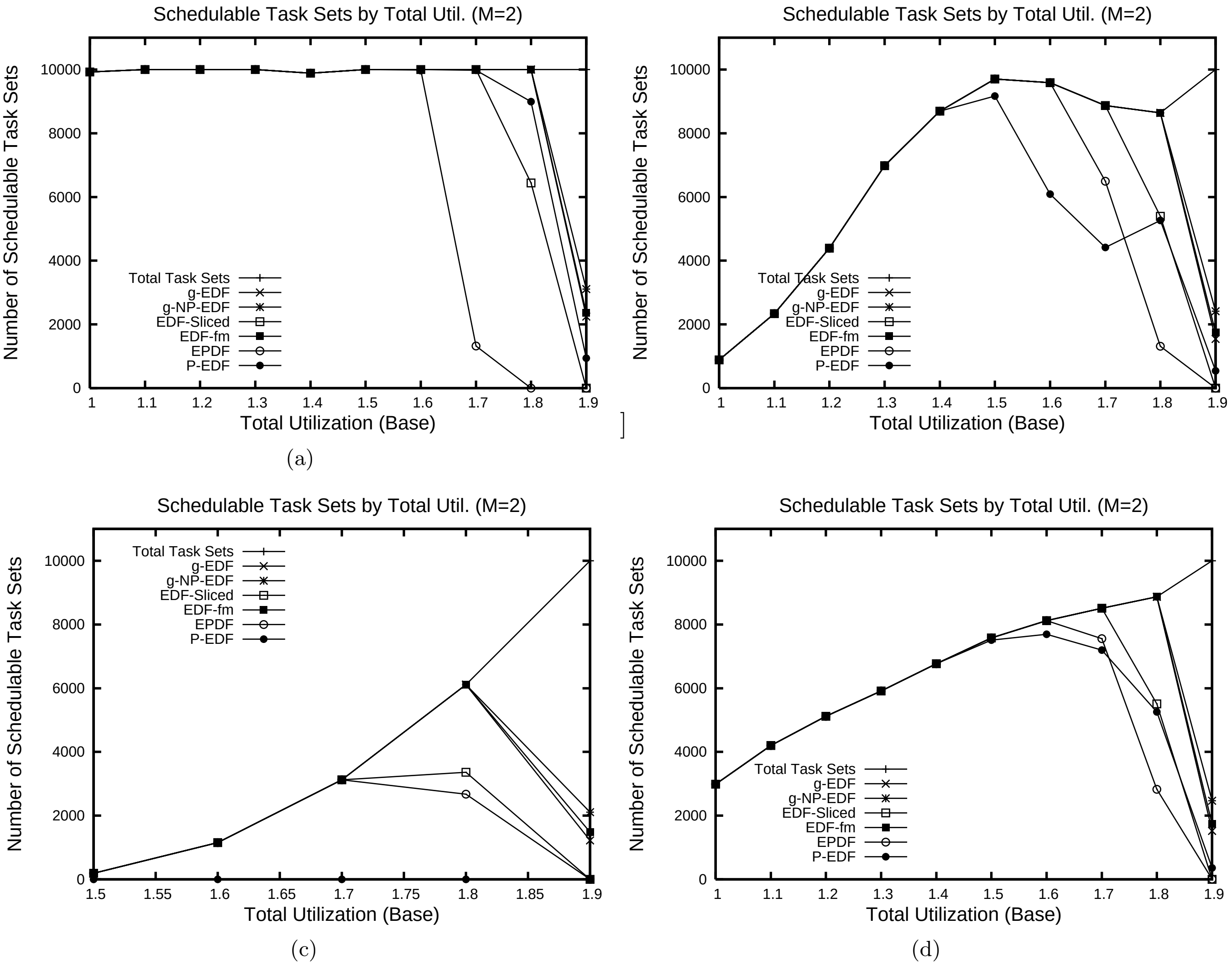


Figure 9.16: Schedulability comparison for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 128K (preemption cost = $116\mu s$, migration cost = $136\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

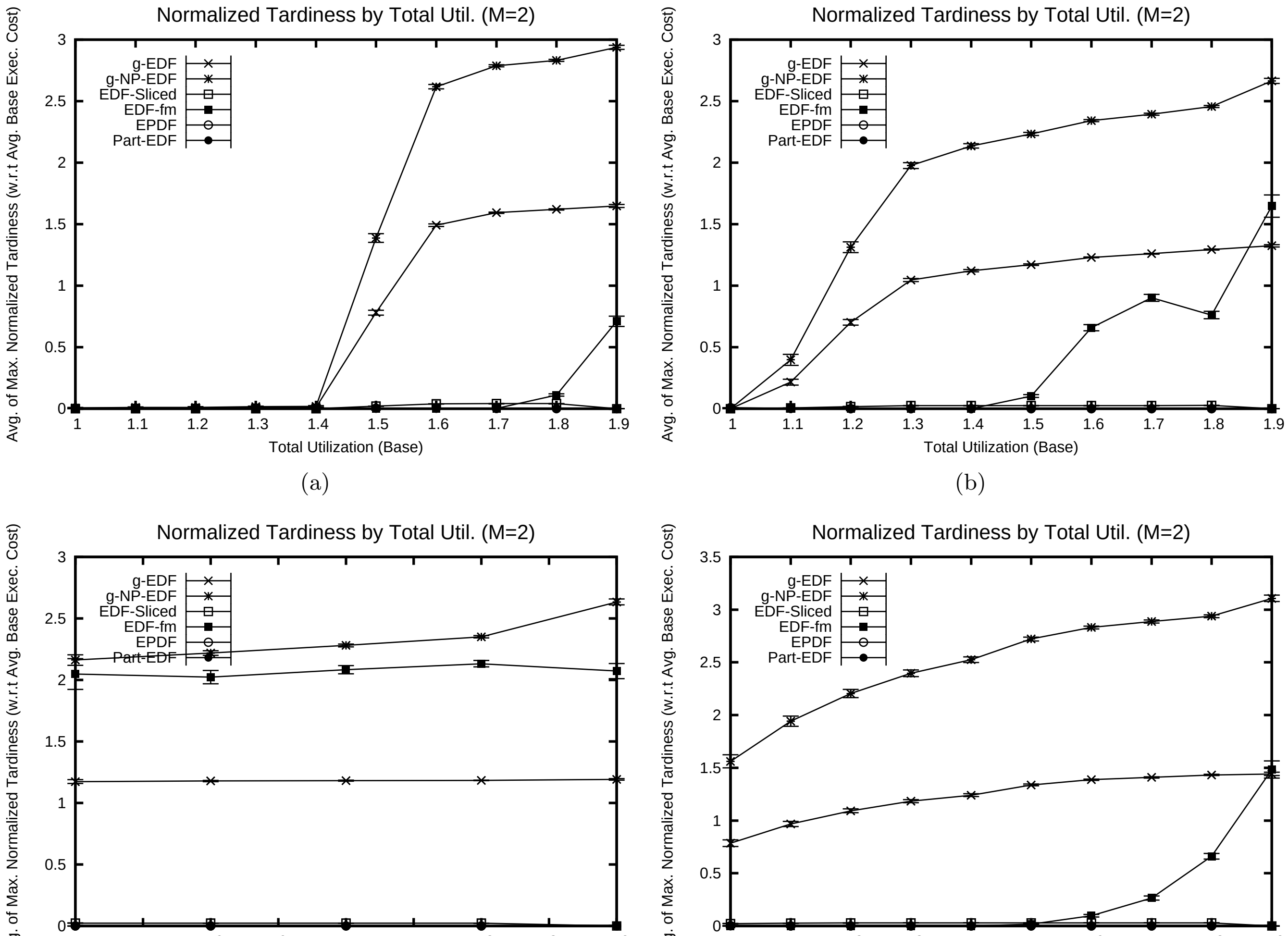


Figure 9.17: Comparison of tardiness bounds for $M = 2$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 128K (preemption cost = $116\mu s$, migration cost = $136\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

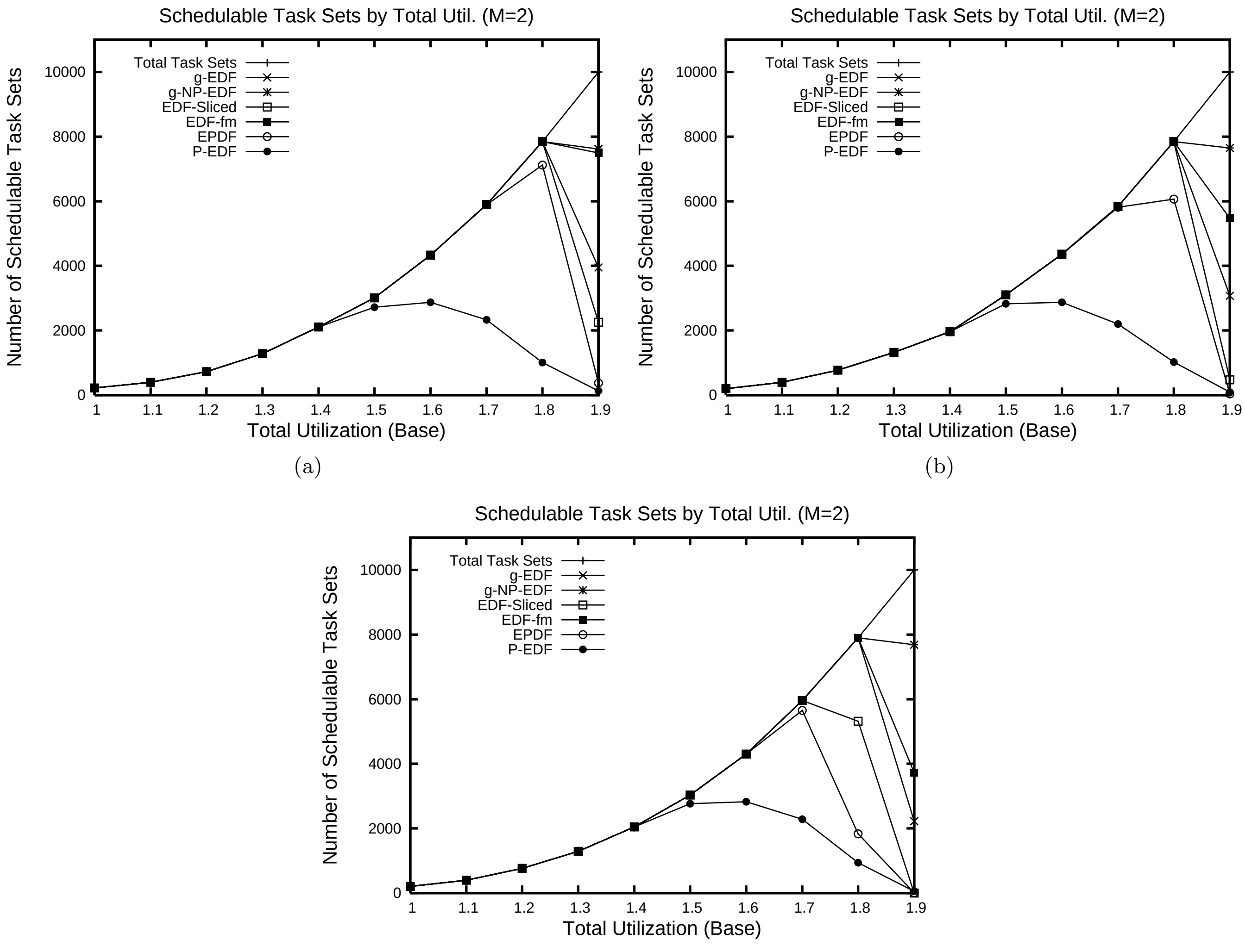


Figure 9.18: Schedulability comparison with task utilizations distributed bimodally between the ranges $[0.1, 0.5)$ and $[0.5, 0.9)$ with probabilities 0.1 and 0.9, respectively, for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, and **(a)** WSS = 4K (preemption cost = $8\mu s$; migration cost = $8\mu s$), **(b)** WSS = 64K (preemption cost = $58\mu s$; migration cost = $64\mu s$), and **(c)** WSS = 128K (preemption cost = $116\mu s$; migration cost = $136\mu s$),

Impact of preemption and migration costs. To study the impact of preemption and migration costs on the various algorithms, we will consider Figures 9.19–9.24, which plot the results for the three preemption and migration costs considered when $M = 4$. Comparing corresponding insets of Figures 9.19, 9.21, and 9.23, which are for WSSs of 4K (preemption cost = $16\mu s$, migration cost = $16\mu s$), 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), respectively, it is evident that EPDF and EDF-sliced are impacted the most by increasing preemption and migration costs. This is in line with the fact that under EPDF and EDF-sliced, each job can be charged with the cost of multiple migrations, whereas under the remaining algorithms, each job is charged with exactly one migration or preemption cost. Considering the remaining algorithms, g-NP-EDF is unaffected as it suffers no preemptions or migrations, and g-EDF is impacted by approximately equal extents at all per-task utilization ranges, whereas the impact to p-EDF and EDF-fm is higher at lower per-task utilizations. This trend observed with p-EDF and EDF-fm is due to the fact that under these algorithms, as task utilizations increase, schedulability is more adversely impacted by difficulties in partitioning tasks among processors than by preemption overheads. Some other trends to observe are as follows. Though g-EDF is impacted by increasing preemption and migration costs, its performance is comparable to that of g-NP-EDF for all WSSs even when total utilization is as high as 90%. The schedulability of g-EDF is higher than that of p-EDF for all WSSs considered, and that of EPDF and EDF-sliced are higher than that of p-EDF when tasks are predominantly heavy, again, for all WSSs).

Comparing the tardiness bounds reported in Figures 9.20, 9.22, and 9.24, trends among the insets of any figure are as described in the context of the impact of task utilization costs. Not much difference is exhibited by corresponding insets of figures associated with different preemption and migration costs.

Impact of M. Barring a few exceptions, varying M while keeping other parameters fixed (or, in the case of preemption and migration costs normalized with respect to M), worsens schedulability equally for all the algorithms. For example, compare corresponding insets of Figures 9.6 and 9.19 or of Figures 9.8 and 9.21. This is due to the assumption that preemption and migration costs increase linearly with increasing M. Therefore, the impact is higher when higher normalized (base) preemption and migration costs are assumed. One significant deviation from this general trend is that the loss incurred by EDF-fm is significantly higher when moving from $M = 2$ to $M = 4$ when task utilizations are high. This is due to the difficulties in task assignment with heavy tasks on more than two processors. In a somewhat similar vein, EPDF and EDF-sliced take a heavy hit when moving from $M = 4$ to $M = 8$ for periods in the range $[10, 100)$ due to high migration costs. It can also be seen that tardiness bounds guaranteed by the non-partitioned EDF algorithms increase with increasing M.

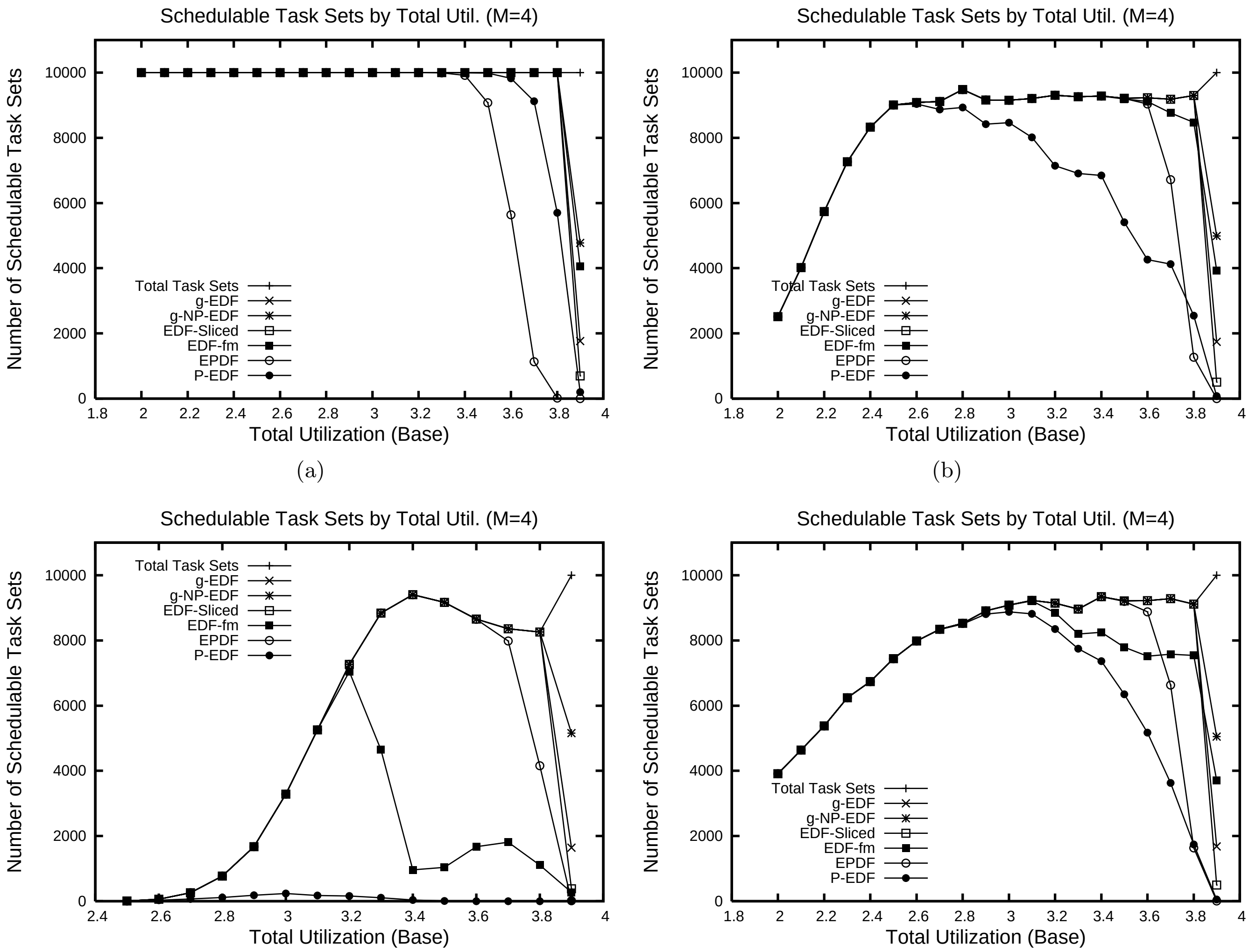
Impact of task periods. As discussed earlier, for algorithms other than EPDF and EDF-sliced, most overheads, with the significant exception of the tick-scheduling overhead, charged per unit of execution cost of a job, decrease with increasing base execution costs (and hence, increasing task periods for a given task utilization). (The tick-scheduling overhead charged per unit of job execution cost is independent of the total execution cost of a job.) Therefore, for these algorithms, in general, the larger the job size, the lower the overheads. A rough estimate of the improvement in schedulability, if any, as the range for periods is increased from $[10, 100)$ to $[100, 500)$ for a given per-task utilization range can be discerned by comparing corresponding insets of any two schedulability figures that differ only in the period parameter. Since preemption and migration costs constitute the predominant overhead, gains are the highest when migration costs are the highest. For example, compare Figures 9.23 and 9.29. $[p_{\min}, p_{\max})$ is $[10, 100)$ for the former and $[100, 500)$ for the latter. Preemption and migration costs are $232\mu s$ and $272\mu s$, respectively, in both these figures. In inset (a) of the first figure, the percentage of schedulable task sets for g-EDF is zero when base total utilization is 3.9, whereas it is close to 20% in the second figure. Similar, though slightly lower, gains can be observed for g-EDF in the other insets also. For p-EDF and EDF-fm, gains are noticeably higher for lower per-task utilizations than for higher per-task utilizations. This trend again, as explained in other contexts, is due to the fact that difficulties in partitioning more adversely impact schedulability than the other overheads as task utilizations increase.

Considering the remaining three algorithms, surprisingly, schedulability improves significantly for EPDF, which can be attributed to a decrease in the rounding overhead with increasing execution cost (regardless of the period). Though schedulability seems to be decreasing for EDF-sliced and g-NP-EDF, this trend is simply an artifact of the bin sizes used on the x-axis. In our schedulability plots, each point along the x-axis spans a utilization range that is 0.1 units wide; for instance, point 3.0 corresponds to the range $[3.0, 3.1)$. With larger periods, the distribution of total utilizations of task sets is skewed toward the higher end of the range, *i.e.*, within a range, more task sets are generated with total base utilizations in the higher end, and hence, more are found to be not schedulable. For instance, if our point of interest along the

x-axis is 3.8 and the associated utilization range is $[3.8, 3.9)$, then the total base utilization is above, say, 3.85, for far more task sets when the task periods are larger than when shorter. This argument is further corroborated by the observation that differences in schedulability can be observed only at the final one or two points in the plot for shorter periods at which schedulability is less than 100%, but significantly higher than 0%. We expect this perceived, apparent difference will be rectified if the bin size along the x-axis is decreased. The reason for the skew in the distribution of utilizations, though, is not very clear.

Impact of the quantum size. The impact of the quantum size Q can be seen by comparing the schedulability plots of Figures 9.19–9.29 with those of Figures 9.45–9.50. Q is 1ms for the figures in the first set and 5ms for those in the second set. The increase in schedulability is the highest for EDF-sliced, with higher gains at higher migration costs, since the number of subjobs, and hence, the migration overhead decreases with increasing quantum size. EDF-sliced also benefits from lower scheduling overhead at higher quantum sizes. Though the number of subtasks decreases for EPDF also with increasing Q, leading to a decrease in migration and scheduling overhead, rounding error increases with increasing Q. At low migration cost (WSS=4K and WSS=64K), in our experiments, the increase in rounding error is not less than the decrease in migration and scheduling overhead, and hence, overall, EPDF incurs a loss. When migration cost is higher (WSS=128K), schedulability increases for EPDF as the decrease in the other overhead more than compensate for the increase in rounding overhead. An increase in schedulability can be observed for the remaining algorithms also. This is due to to the decrease in tick-scheduling overhead with increasing quantum sizes.

To summarize, our findings are as follows. These findings are subject to our assumptions holding and are with respect to the task sets generated. First, schedulability is highest under g-NP-EDF; however, tardiness bounds are also higher under it than under the other algorithms. Hence, g-NP-EDF may be used if no other algorithm is capable of scheduling the system at hand and if the tardiness bound that it can guarantee is tolerable. Second, unless migration costs and total utilization are both high, in most cases, g-EDF closely trails g-NP-EDF in terms of schedulability while guaranteeing a tardiness bound that is lower than that of g-NP-EDF by approximately a value equal to the average execution cost. Similarly, when task utilizations are low, EDF-fm exhibits good schedulability and guarantees lower tardiness (sometimes by a significant amount) than both g-EDF and g-NP-EDF. Finally, except when the per-task utilization range or the total base utilization is low, schedulability is somewhat low under



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

Figure 9.19: Schedulability comparison for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $16\mu s$, and migration cost = $16\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

the remaining three algorithms (EPDF, EDF-sliced, and p-EDF). However, tardiness is either zero or minimal under these algorithms for schedulable task systems. Among these three algorithms, p-EDF performs better than EPDF and EDF-sliced (in terms of schedulability) if task utilizations are low and the total utilization is approximately 90%, while at higher per-task utilizations, EPDF and EDF-sliced fare much better. One of these algorithms may be preferable for systems with low per-task and / or total utilizations.

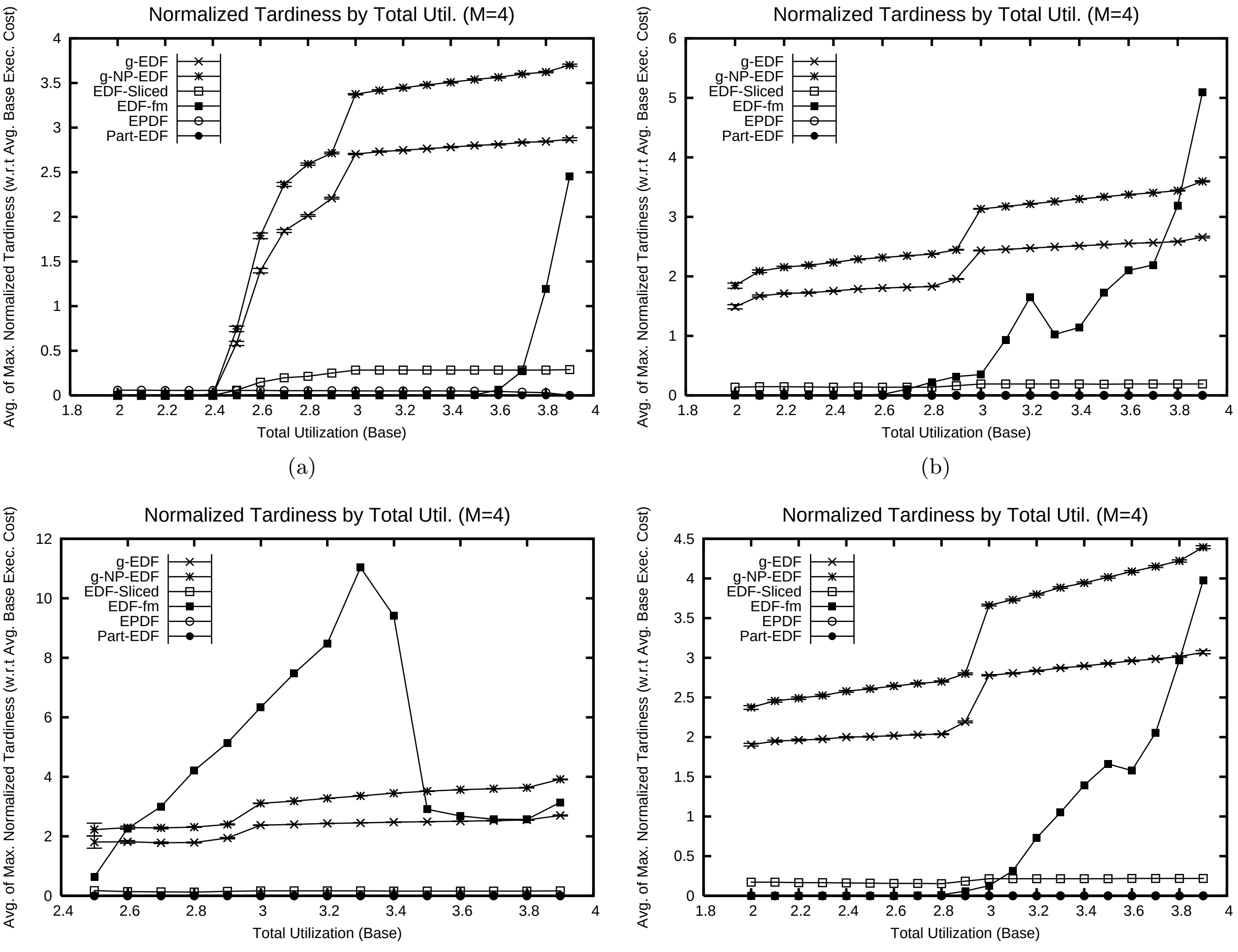


Figure 9.20: Comparison of tardiness bounds for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $16\mu s$, migration cost = $16\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

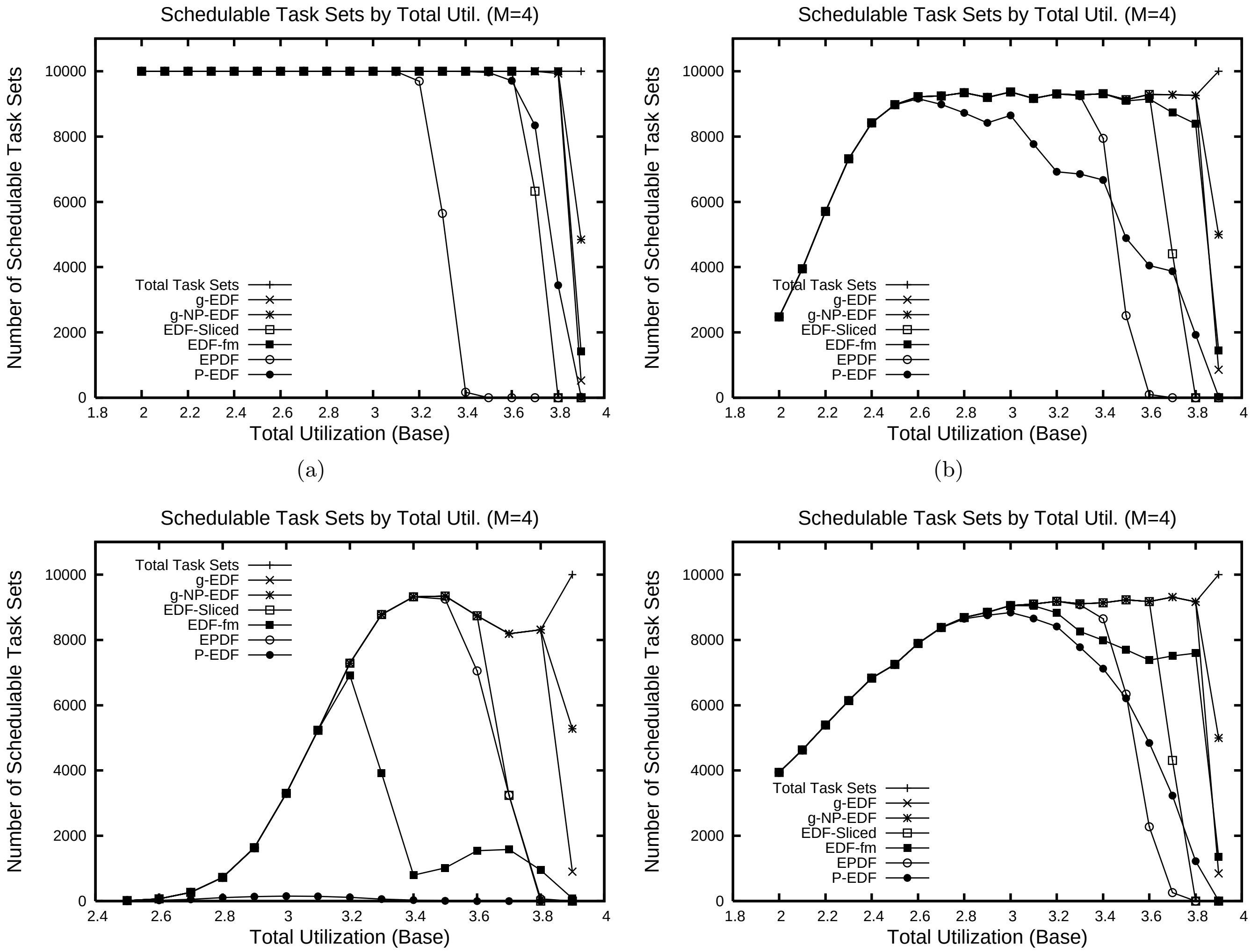


Figure 9.21: Schedulability comparison for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

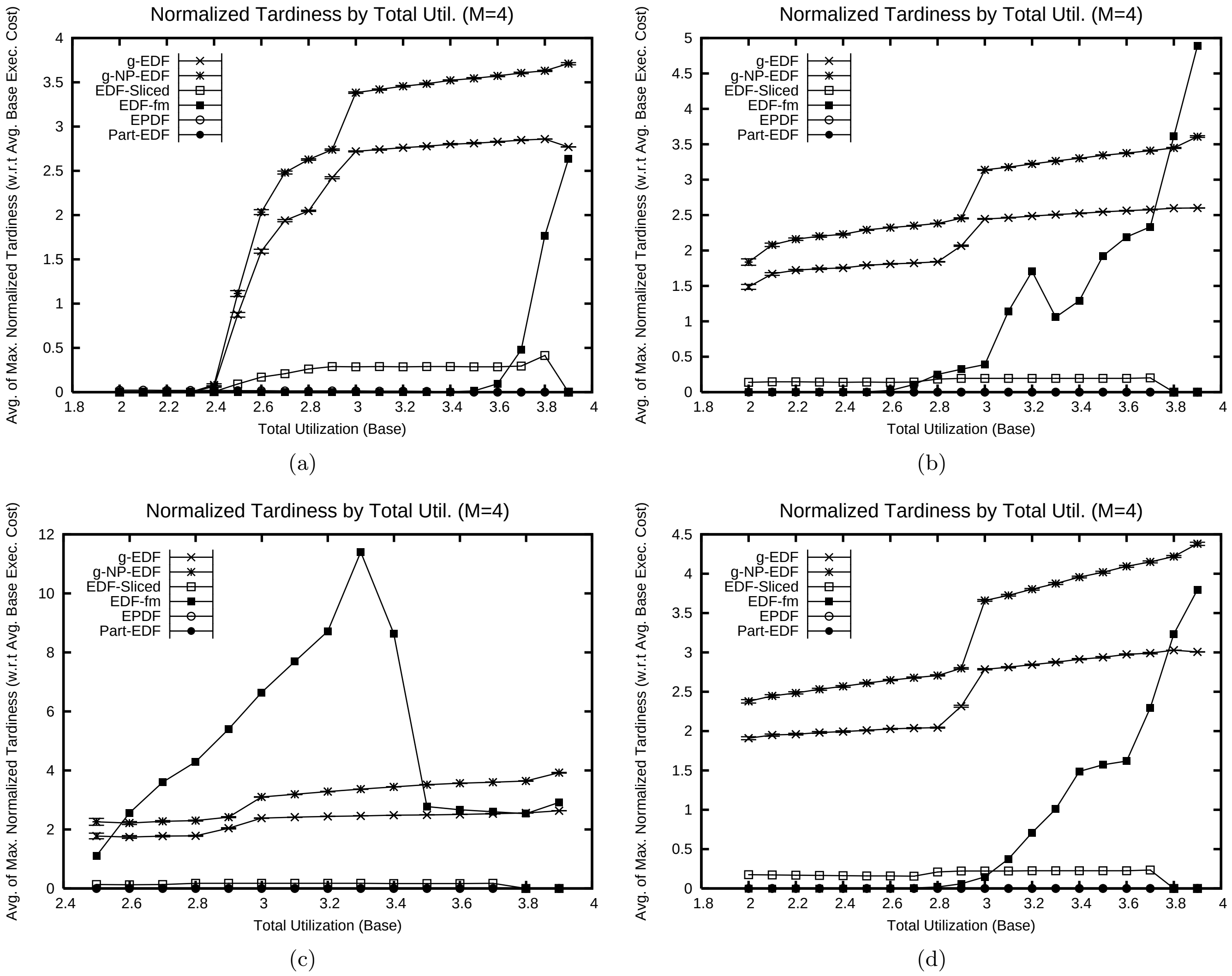


Figure 9.22: Comparison of tardiness bounds for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

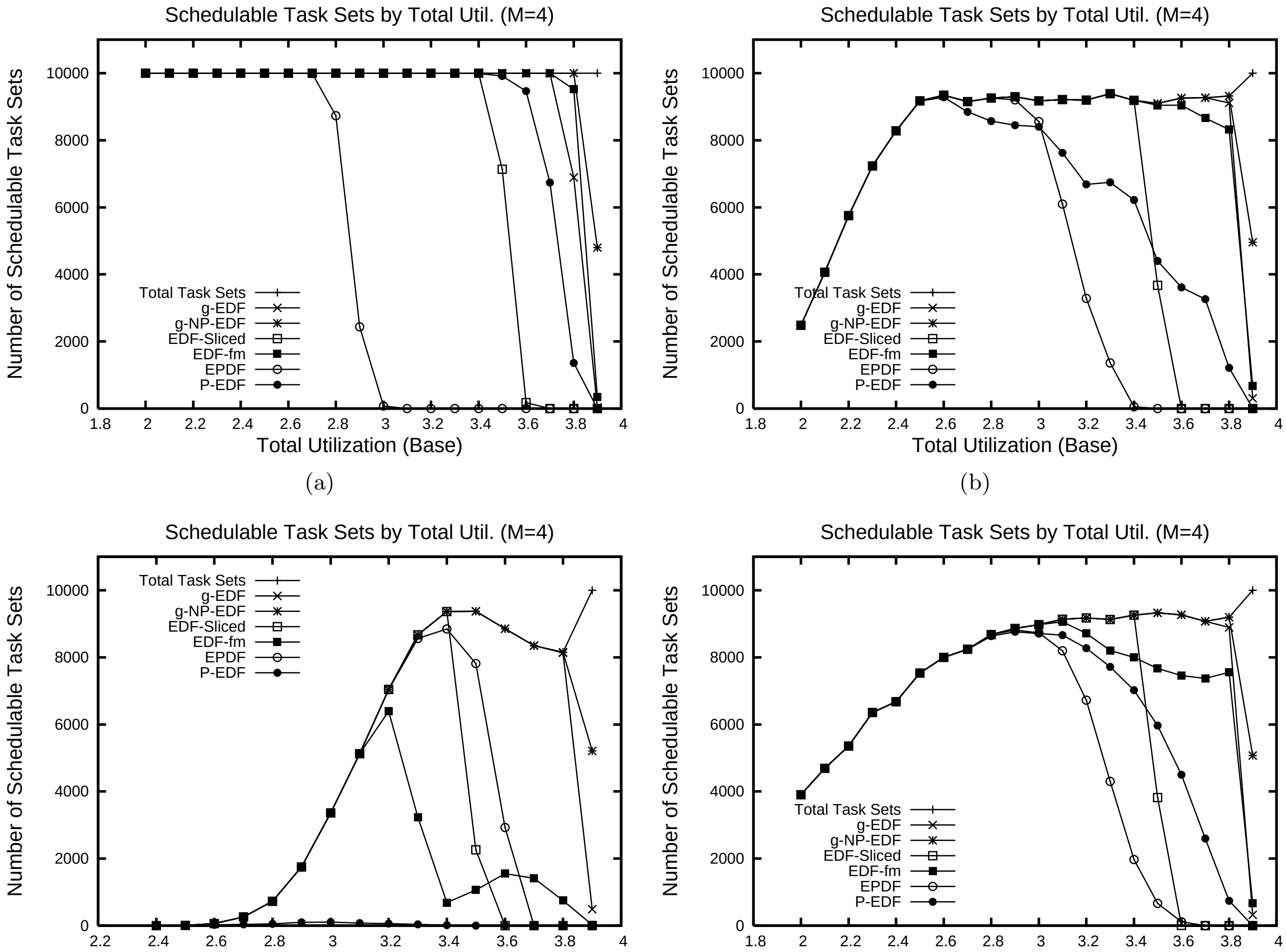


Figure 9.23: Schedulability comparison for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

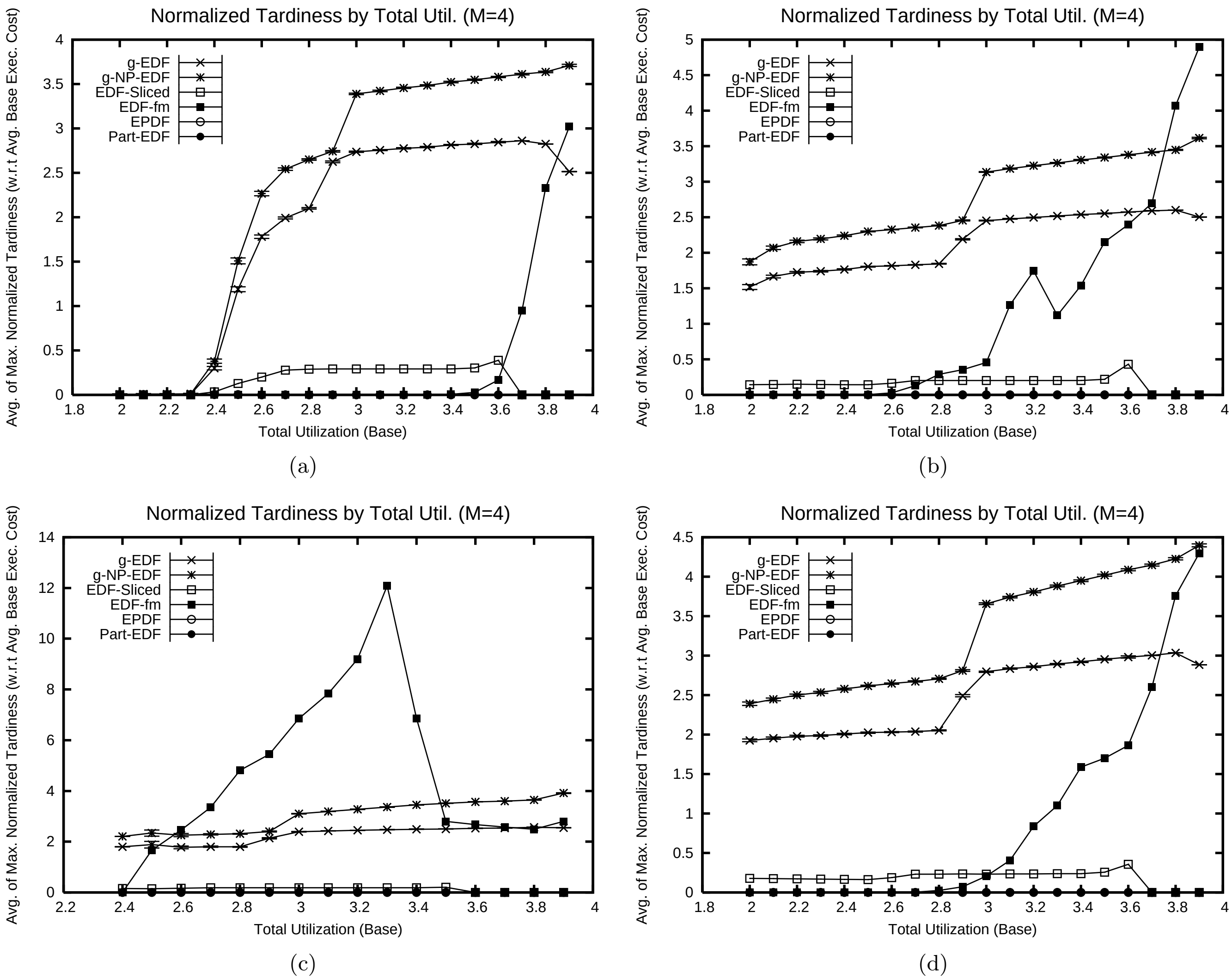


Figure 9.24: Tardiness bounds results for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K Comparison of tardiness bounds for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

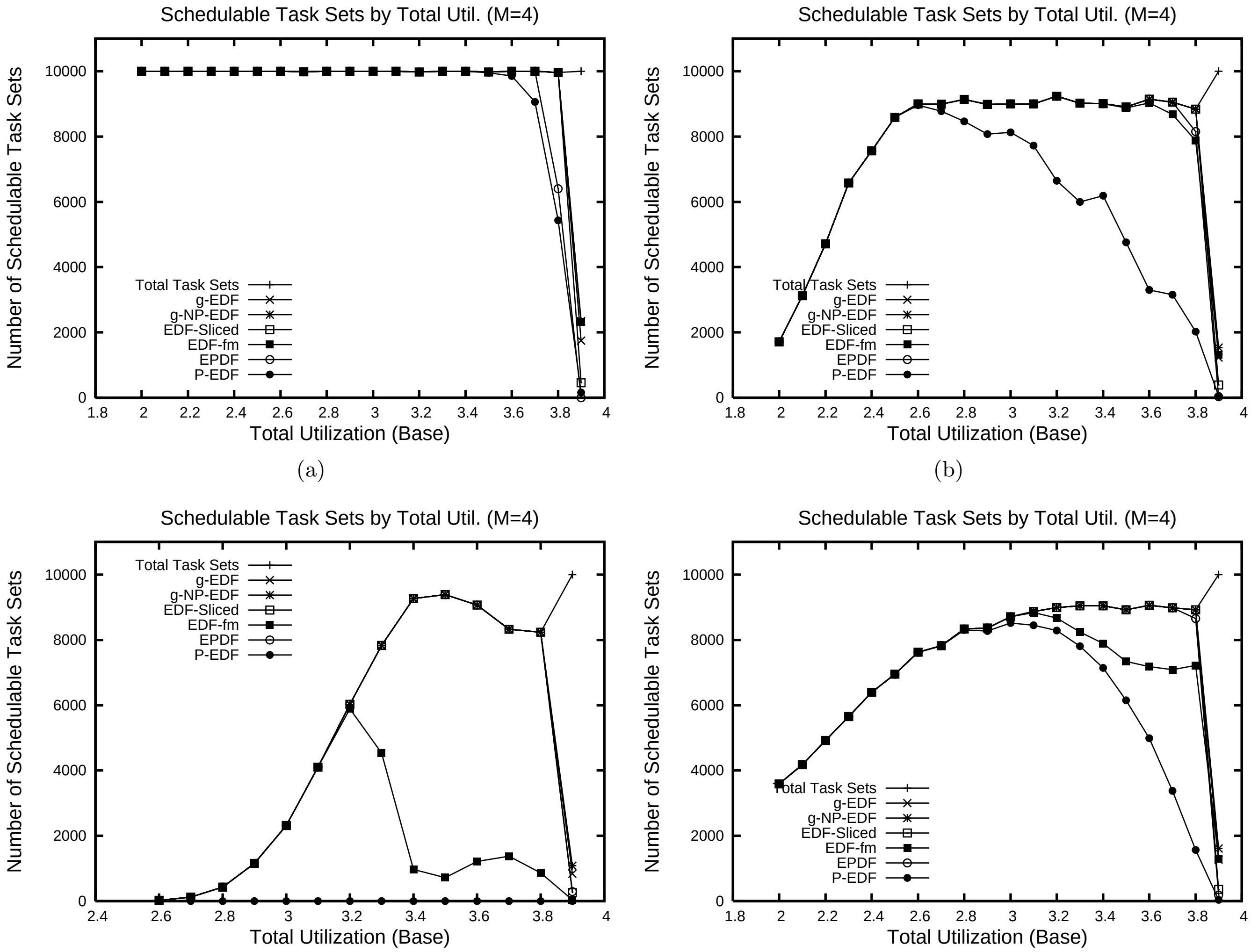


Figure 9.25: Schedulability comparison for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $16\mu s$, and migration cost = $16\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

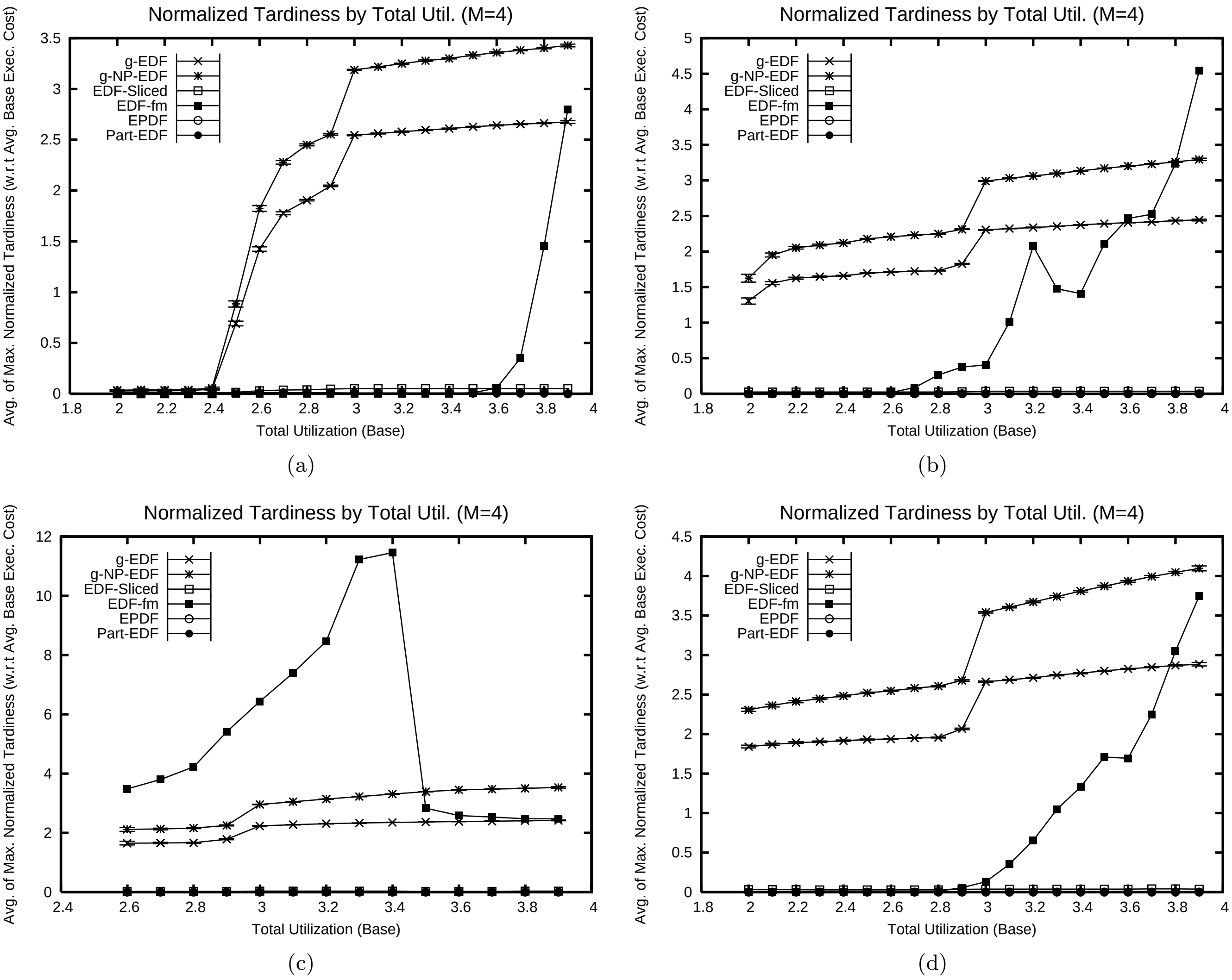


Figure 9.26: Comparison of tardiness bounds for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $16\mu s$, migration cost = $16\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

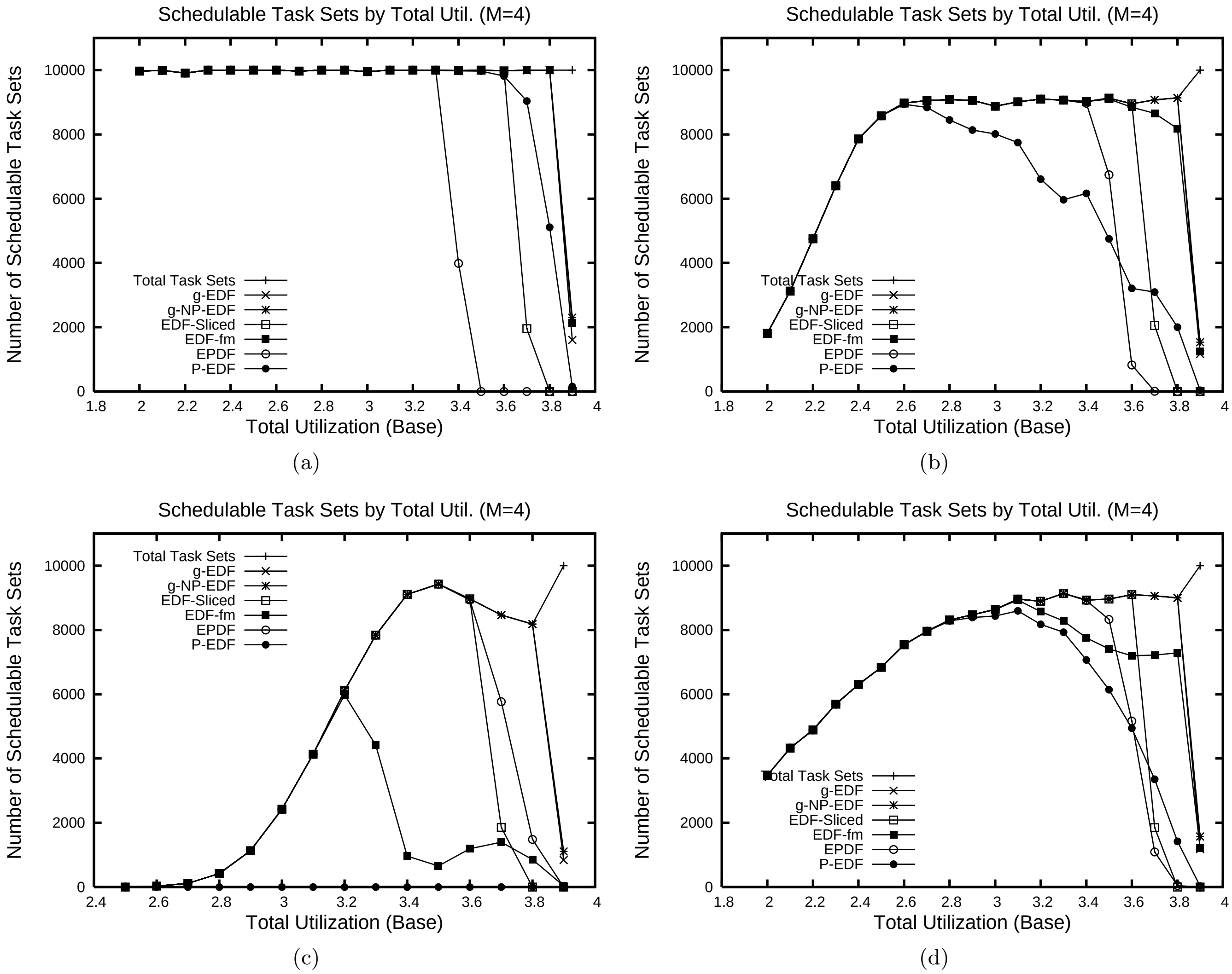
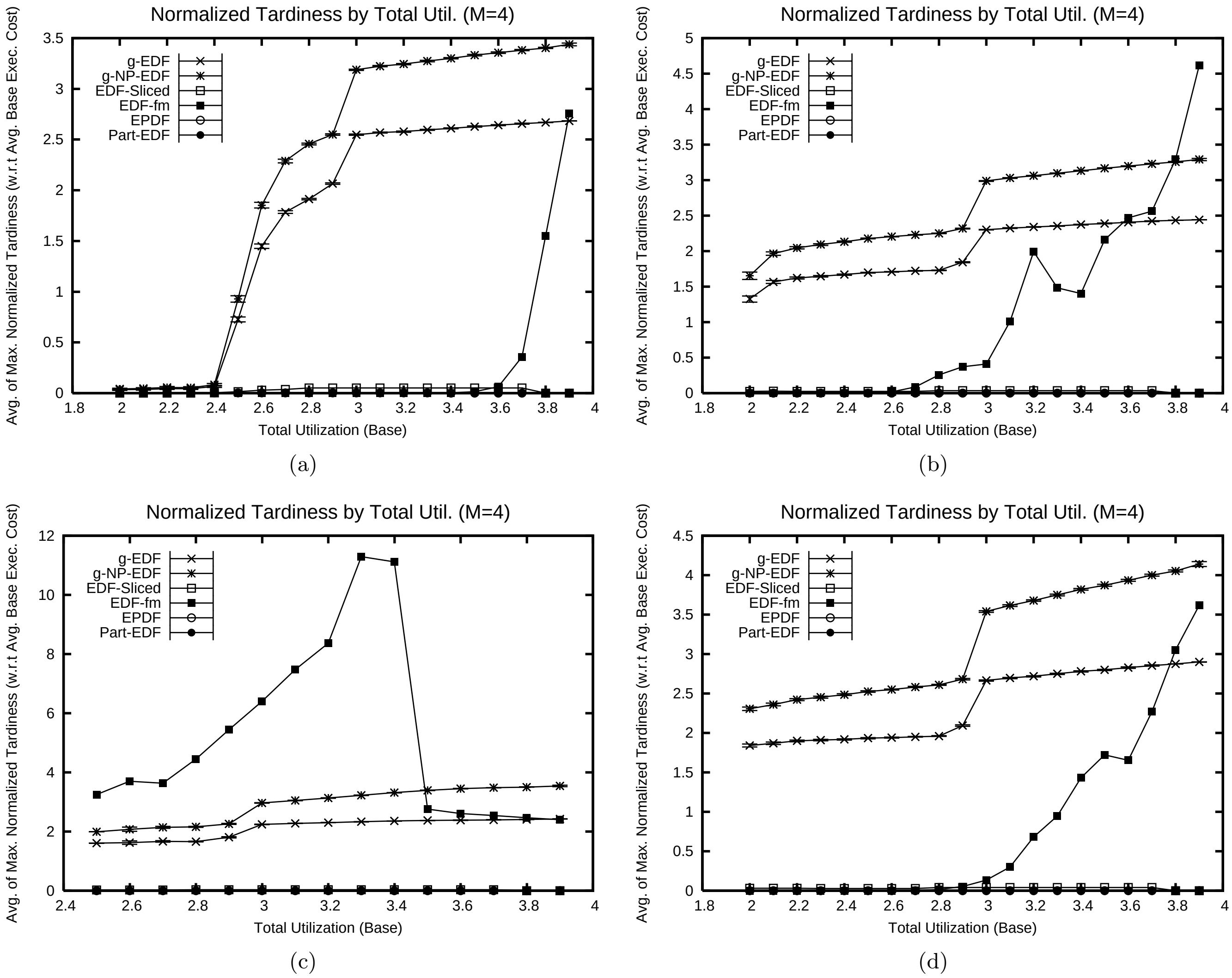


Figure 9.27: Schedulability comparison for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.



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

Figure 9.28: Comparison of tardiness bounds for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

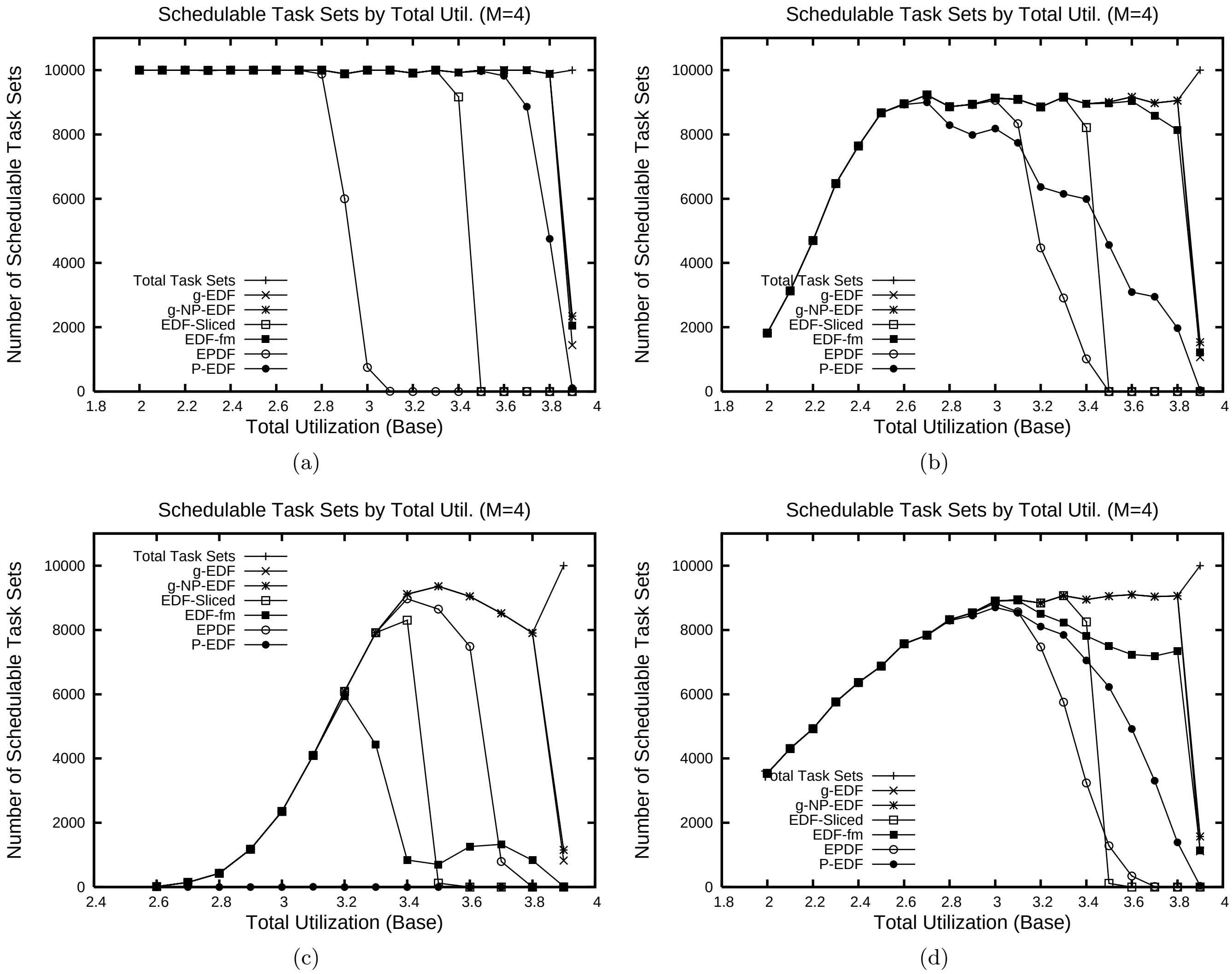


Figure 9.29: Schedulability comparison for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

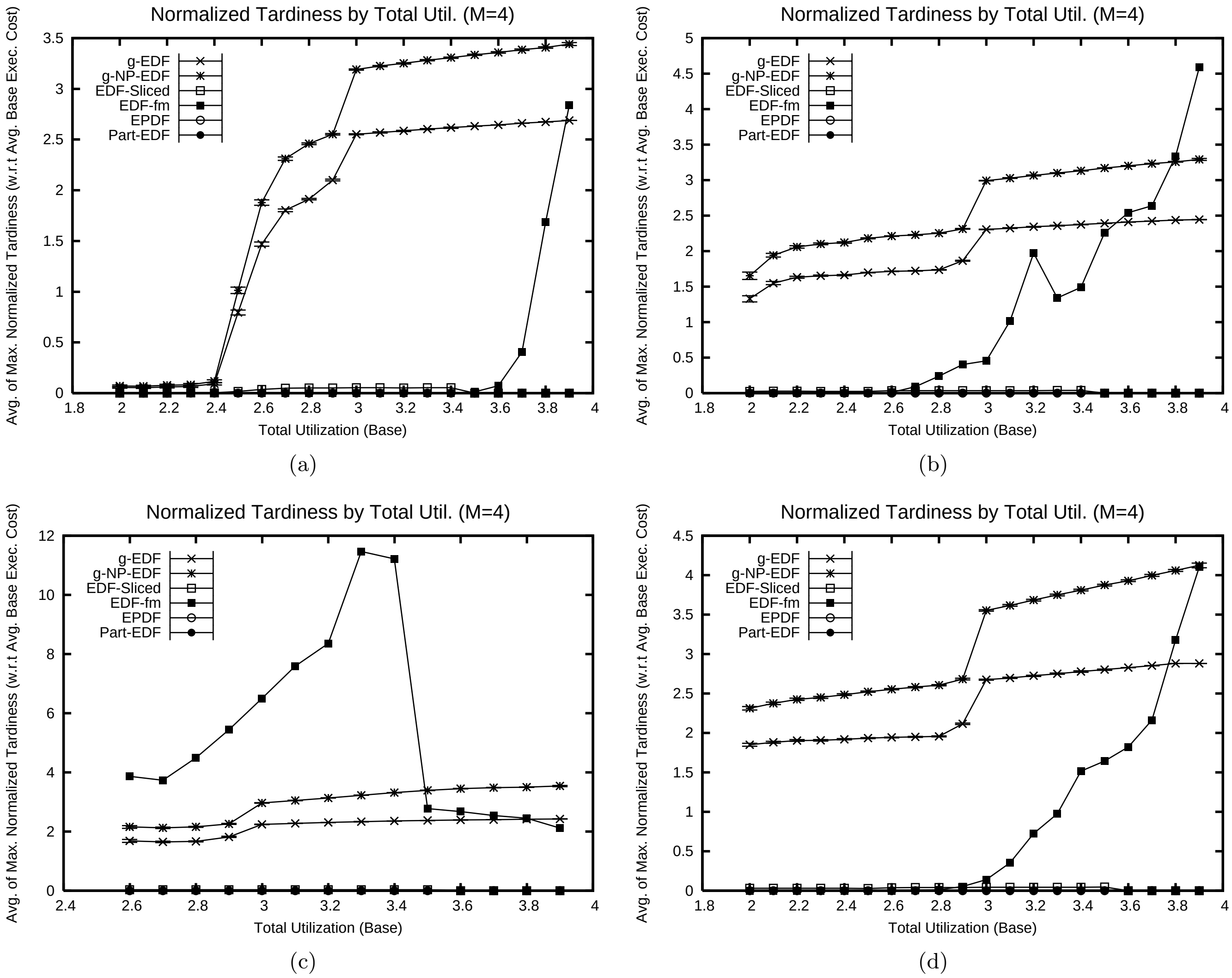


Figure 9.30: Comparison of tardiness bounds for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

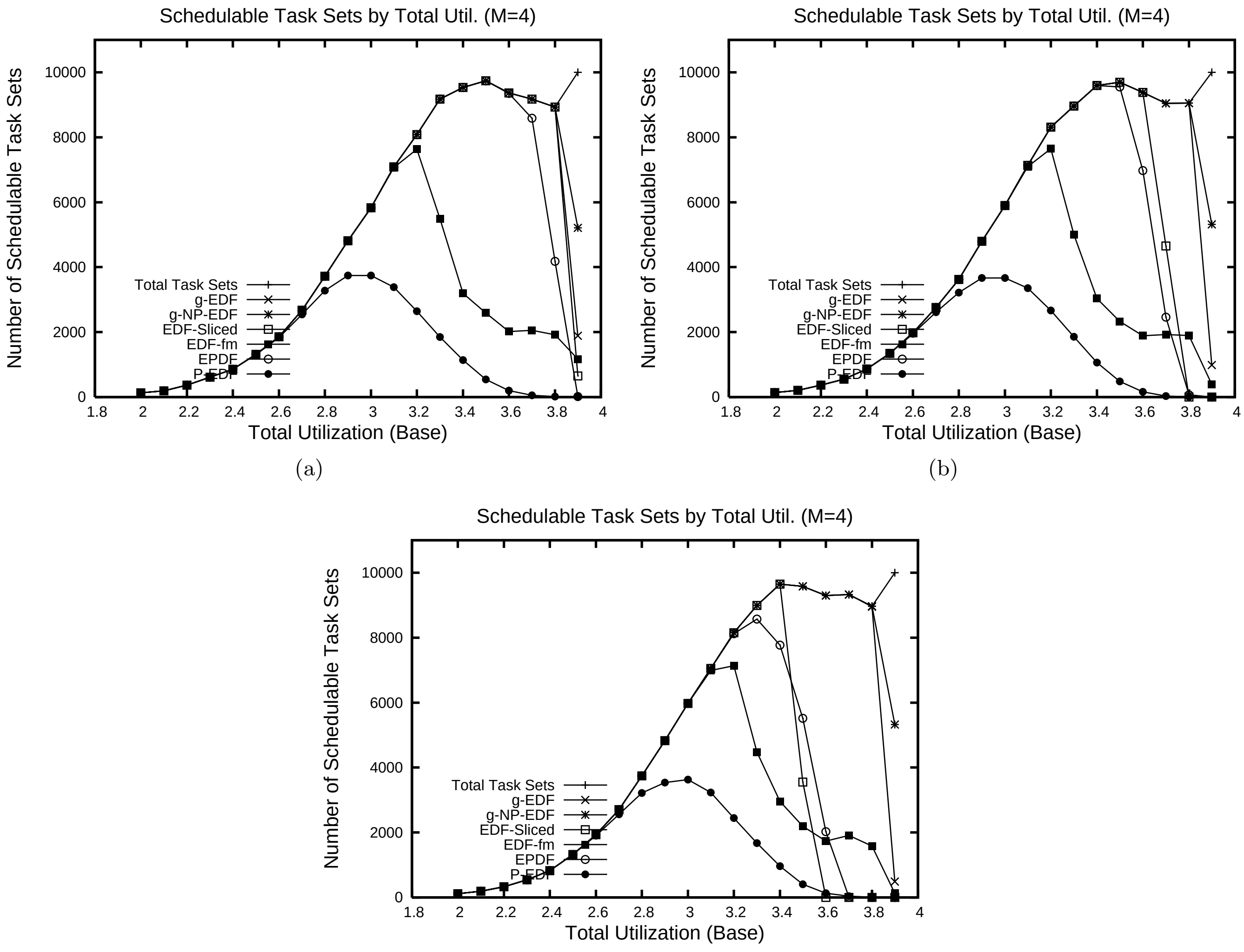
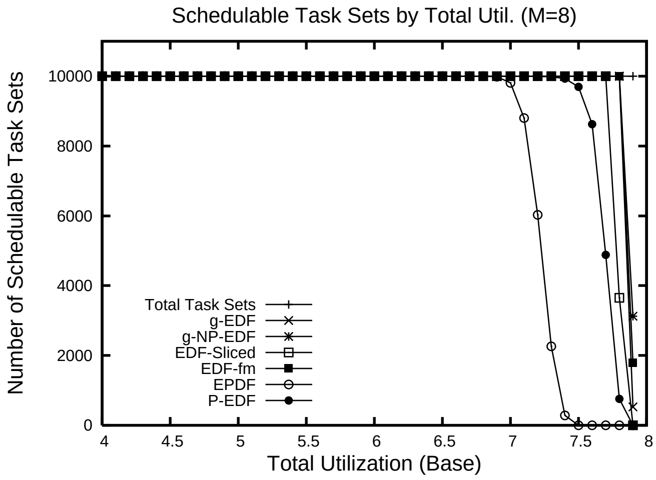
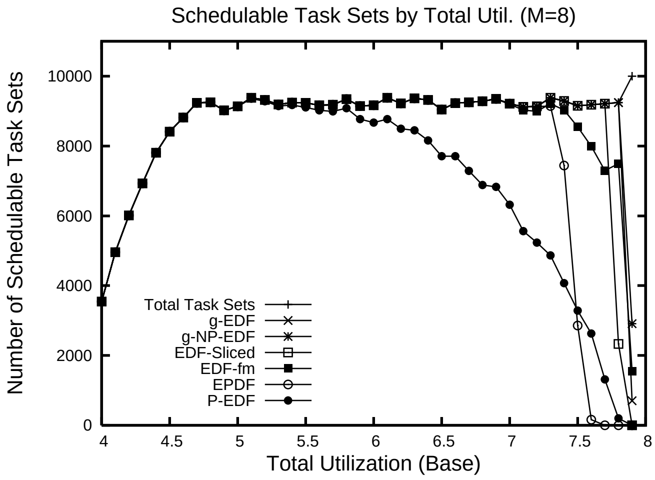


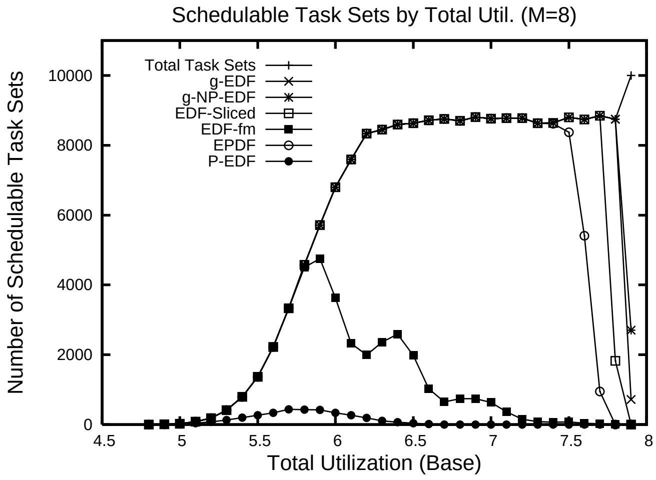
Figure 9.31: Schedulability comparison with task utilizations distributed bimodally between the ranges $[0.1, 0.5)$ and $[0.5, 0.9)$ with probabilities 0.1 and 0.9, respectively, for $M = 4$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, and **(a)** WSS = 4K (preemption cost = $16\mu s$; migration cost = $16\mu s$), **(b)** WSS = 64K (preemption cost = $116\mu s$; migration cost = $128\mu s$), and **(c)** WSS = 128K (preemption cost = $232\mu s$; migration cost = $272\mu s$),



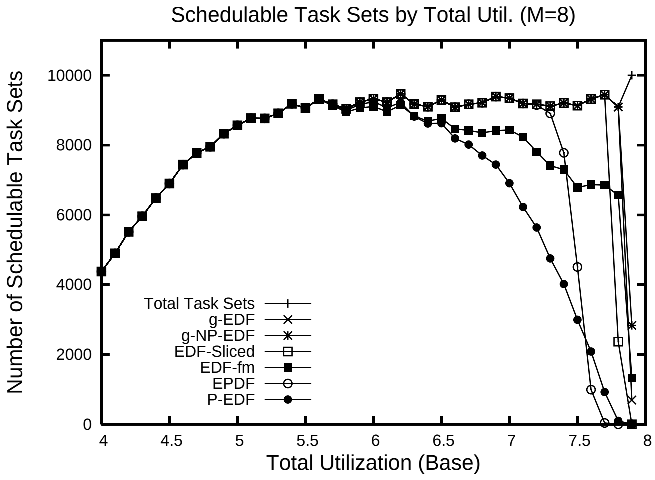
(a)



(b)



(c)



(d)

Figure 9.32: Schedulability comparison for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $32\mu s$, and migration cost = $32\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

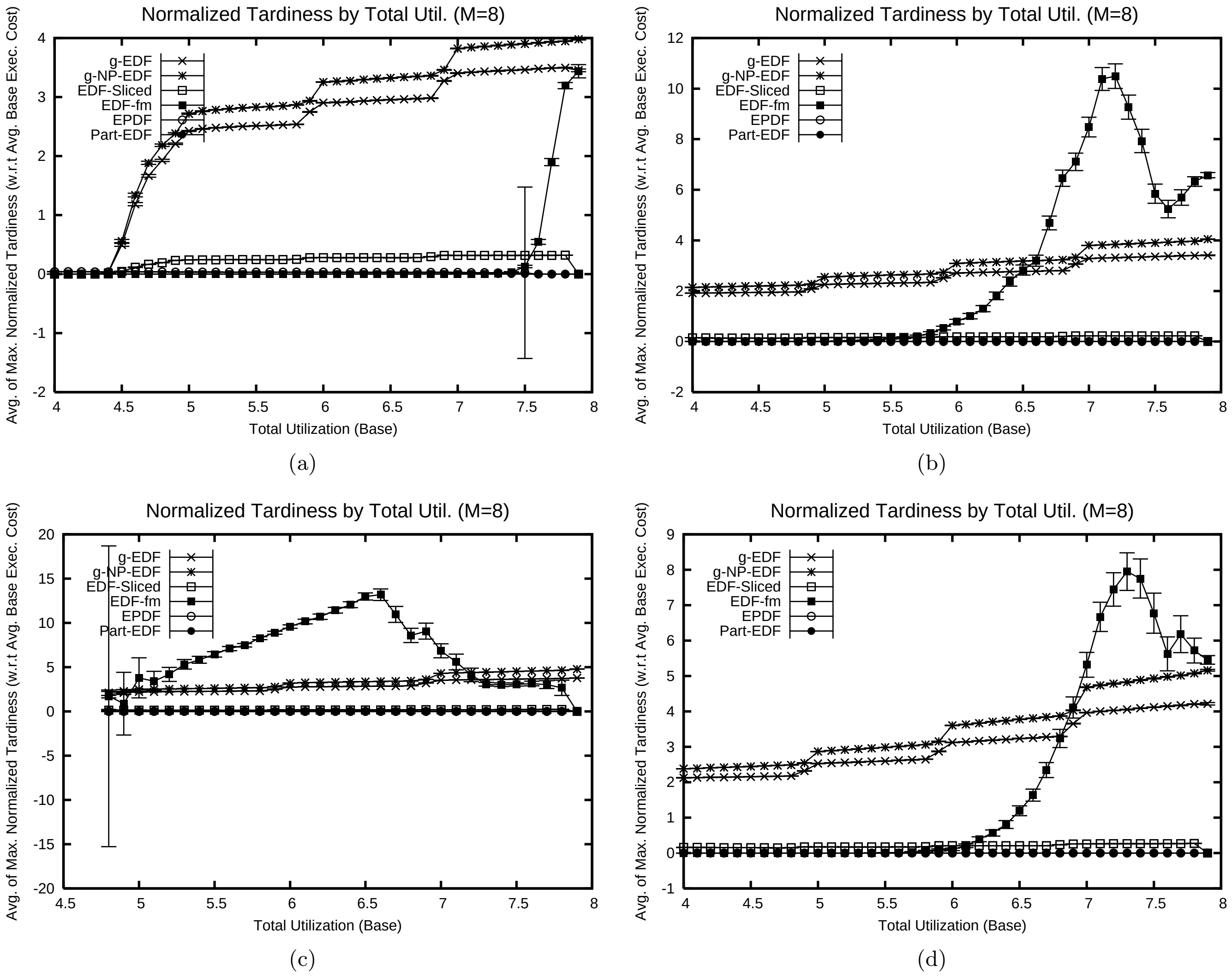


Figure 9.33: Comparison of tardiness bounds for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $32\mu s$, migration cost = $32\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

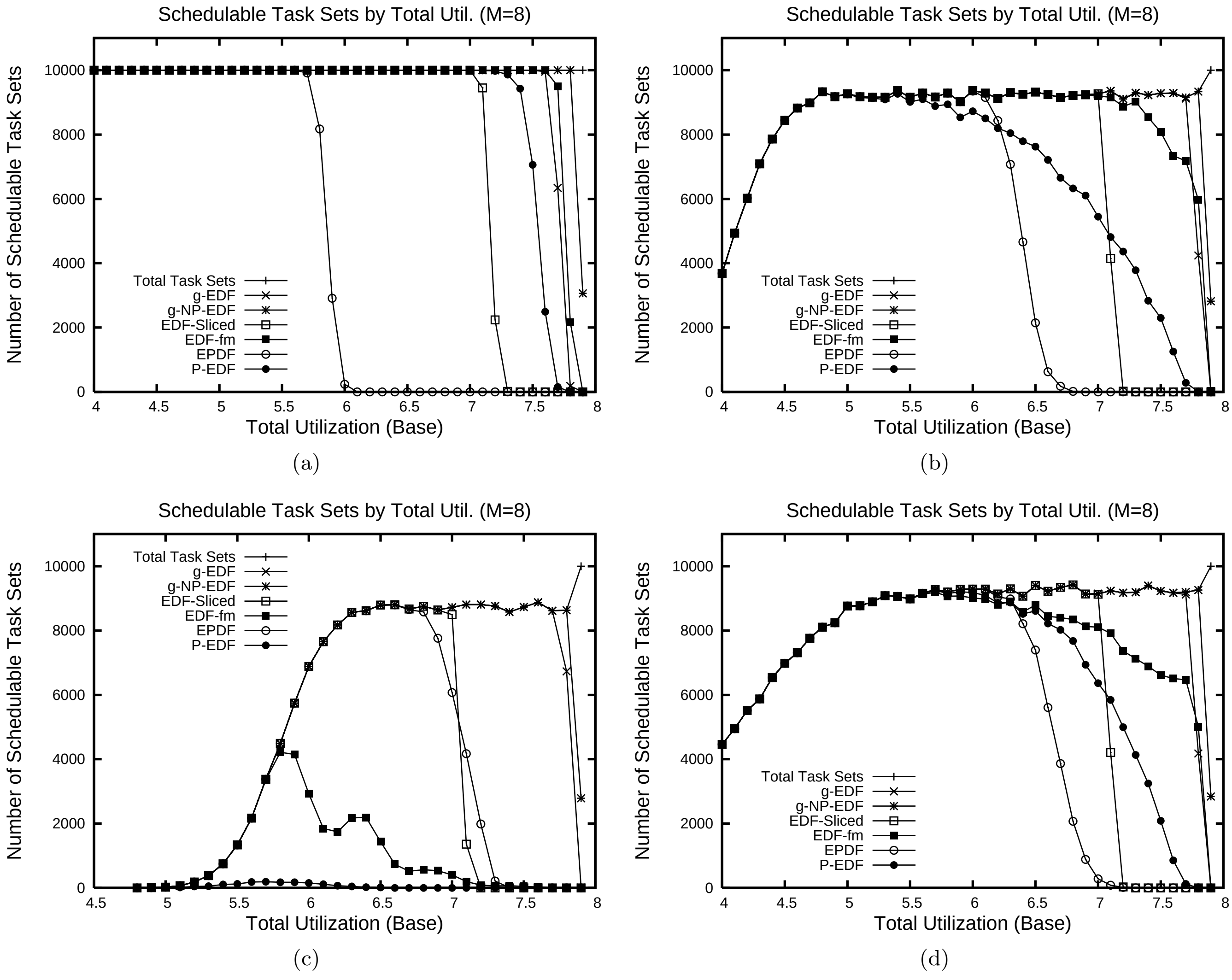
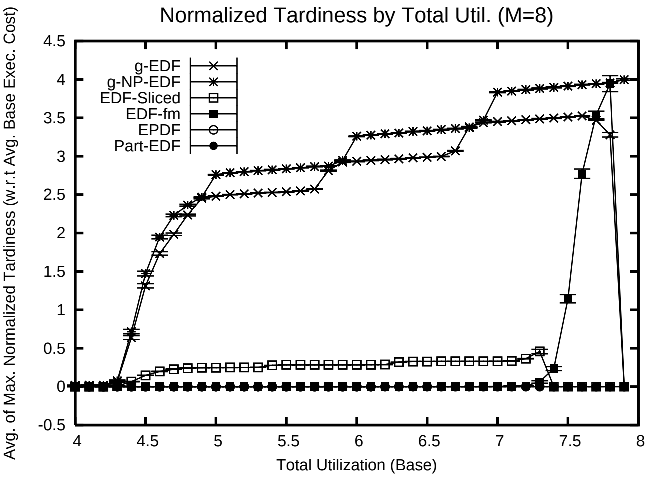
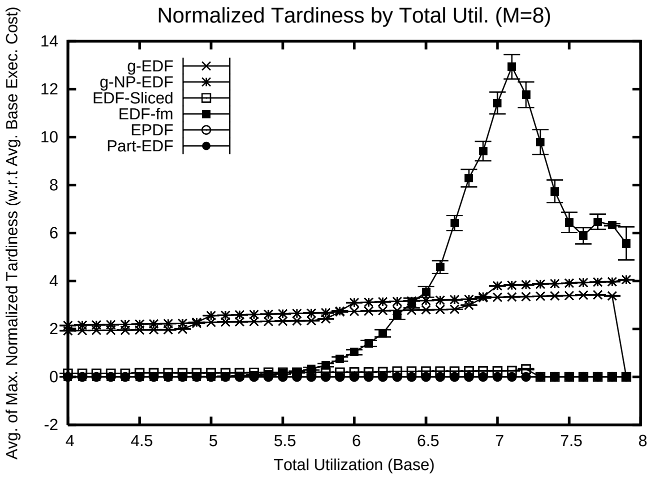


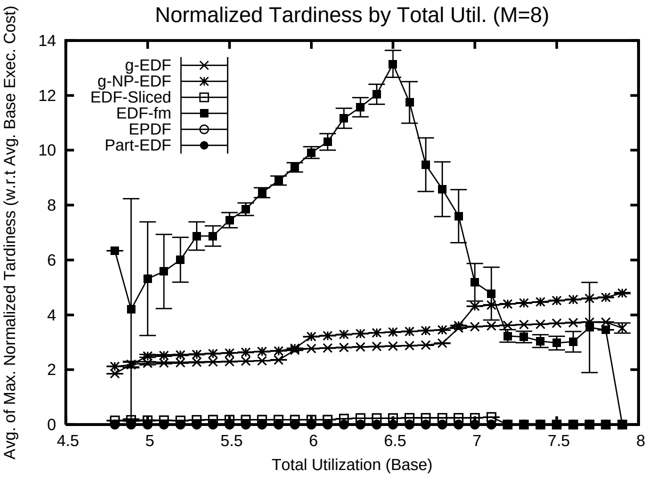
Figure 9.34: Schedulability comparison for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 64K (preemption cost = $232\mu s$, migration cost = $256\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.



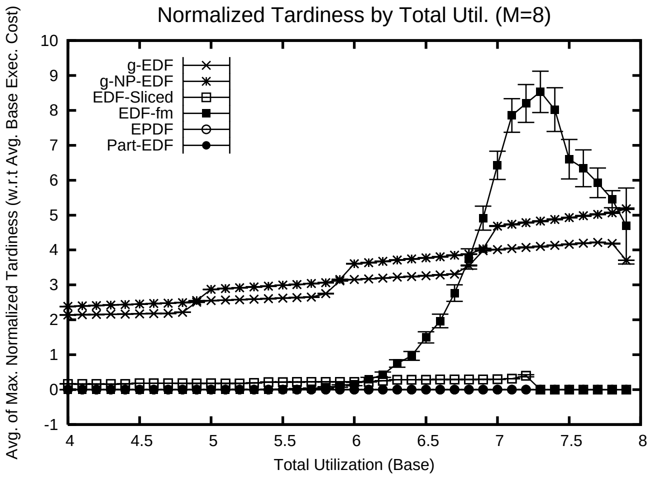
(a)



(b)



(c)



(d)

Figure 9.35: Comparison of tardiness bounds for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 64K (preemption cost = $232\mu s$, migration cost = $256\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

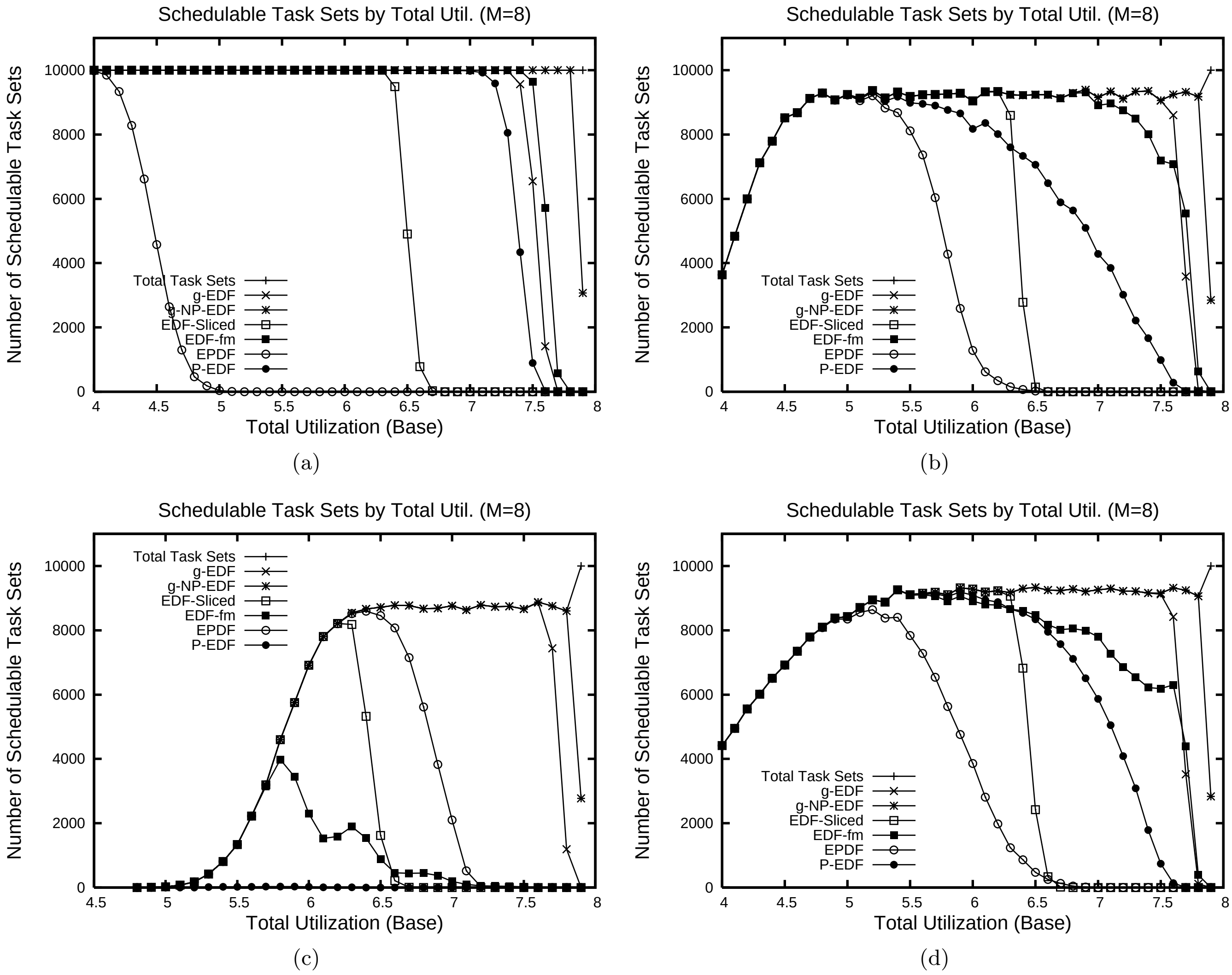


Figure 9.36: Schedulability comparison for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $464\mu s$, migration cost = $544\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

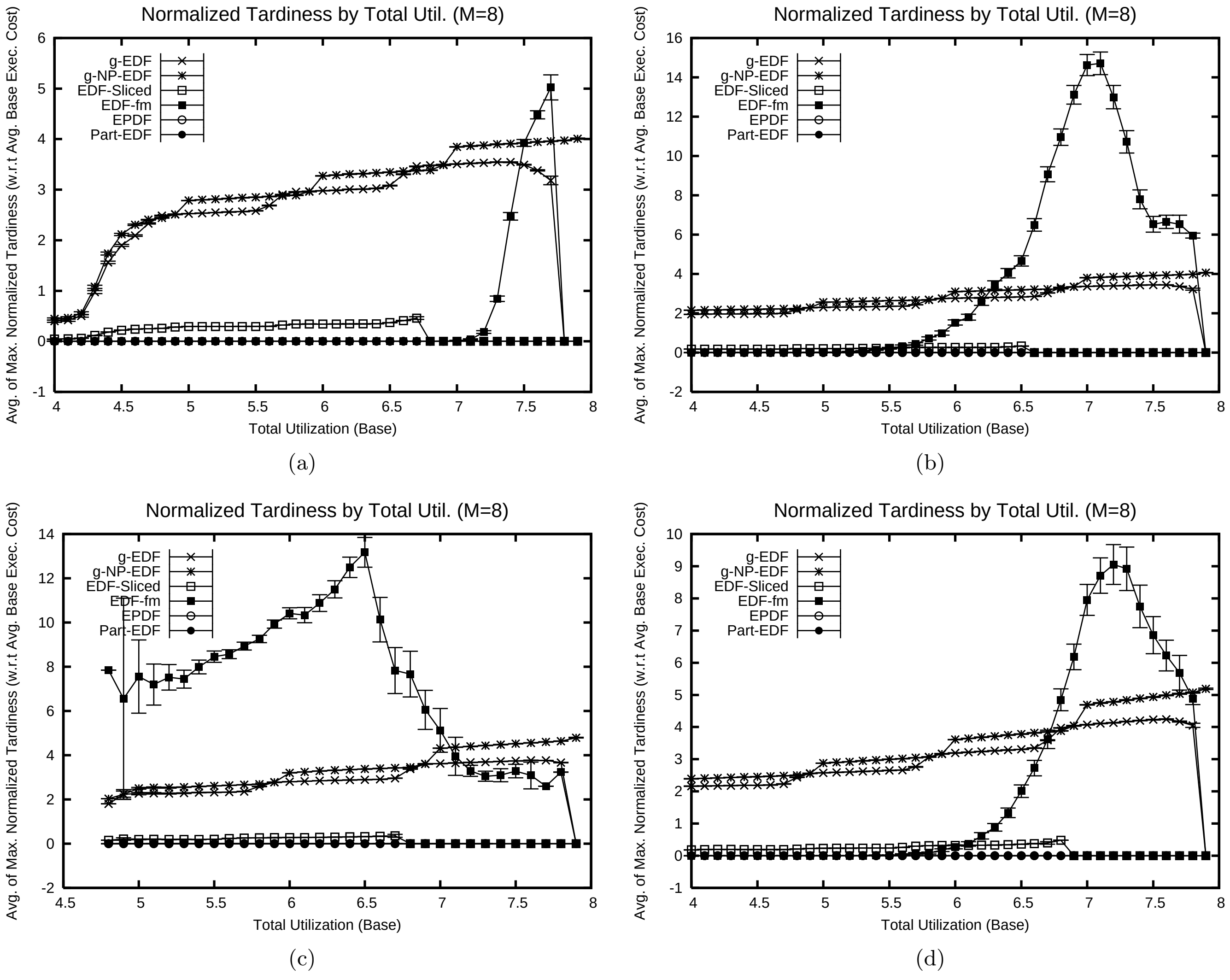


Figure 9.37: Comparison of tardiness bounds for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $464\mu s$, migration cost = $544\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

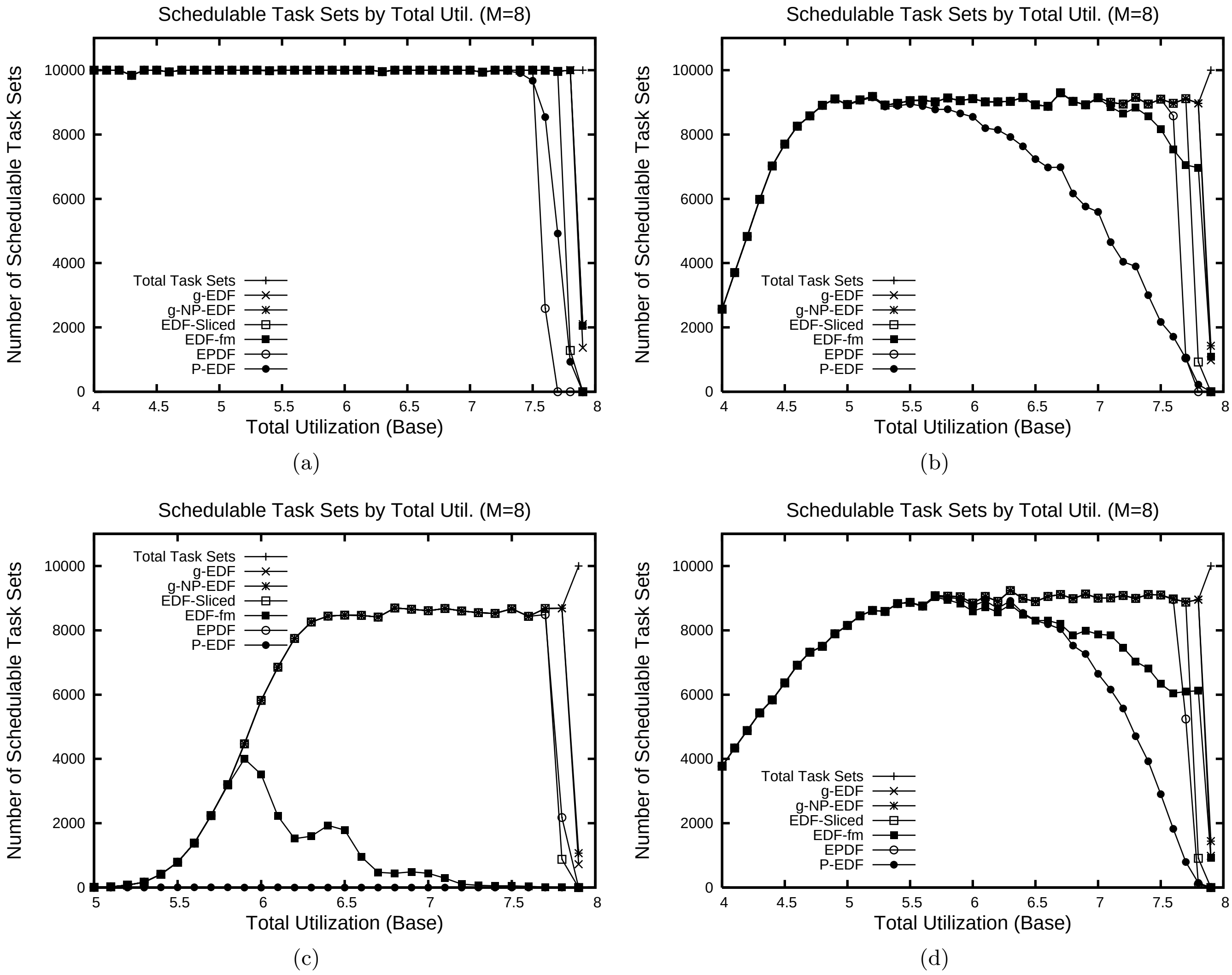


Figure 9.38: Schedulability comparison for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $32\mu s$, and migration cost = $32\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

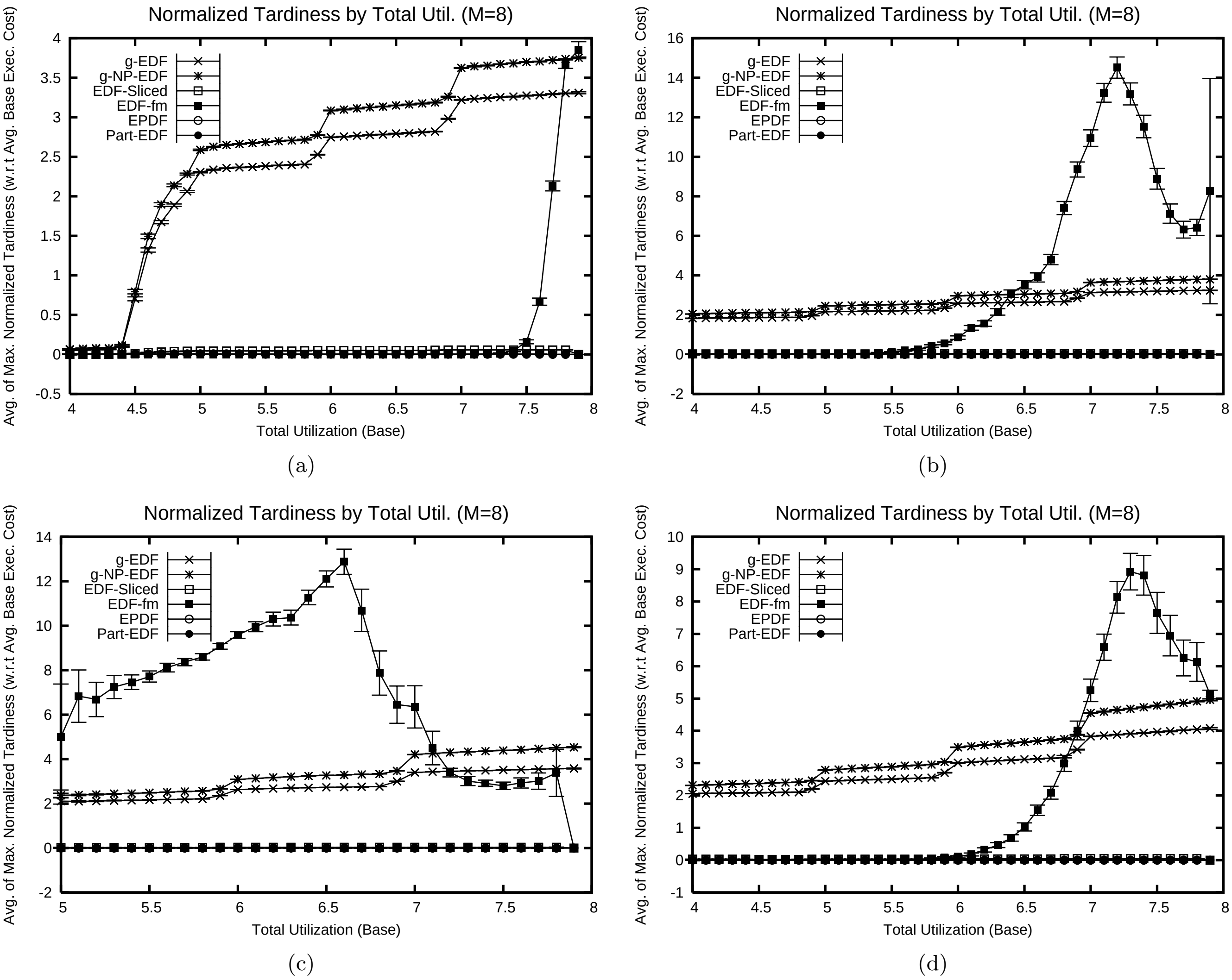


Figure 9.39: Comparison of tardiness bounds for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $32\mu s$, migration cost = $32\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

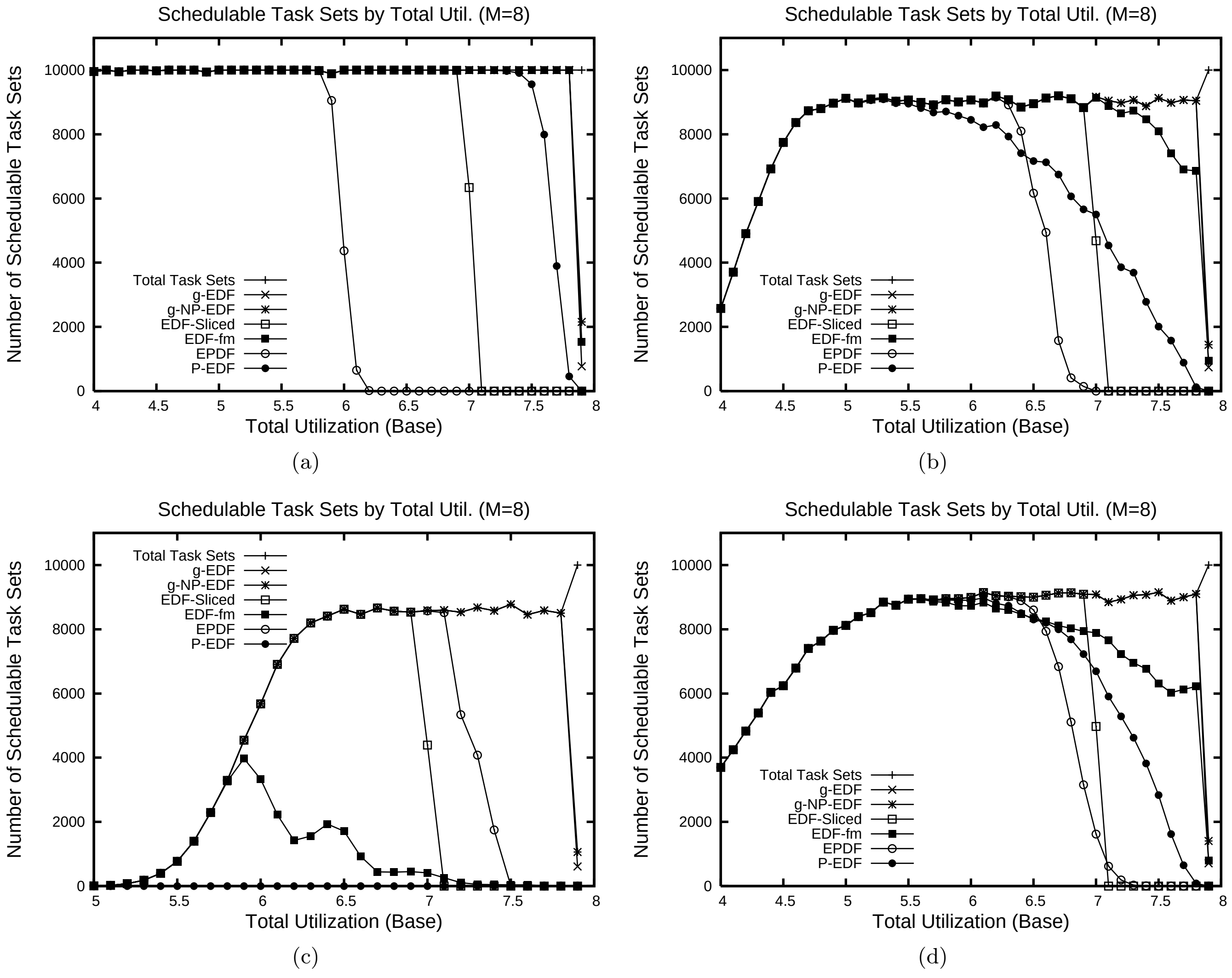


Figure 9.40: Schedulability comparison for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $232\mu s$, migration cost = $256\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

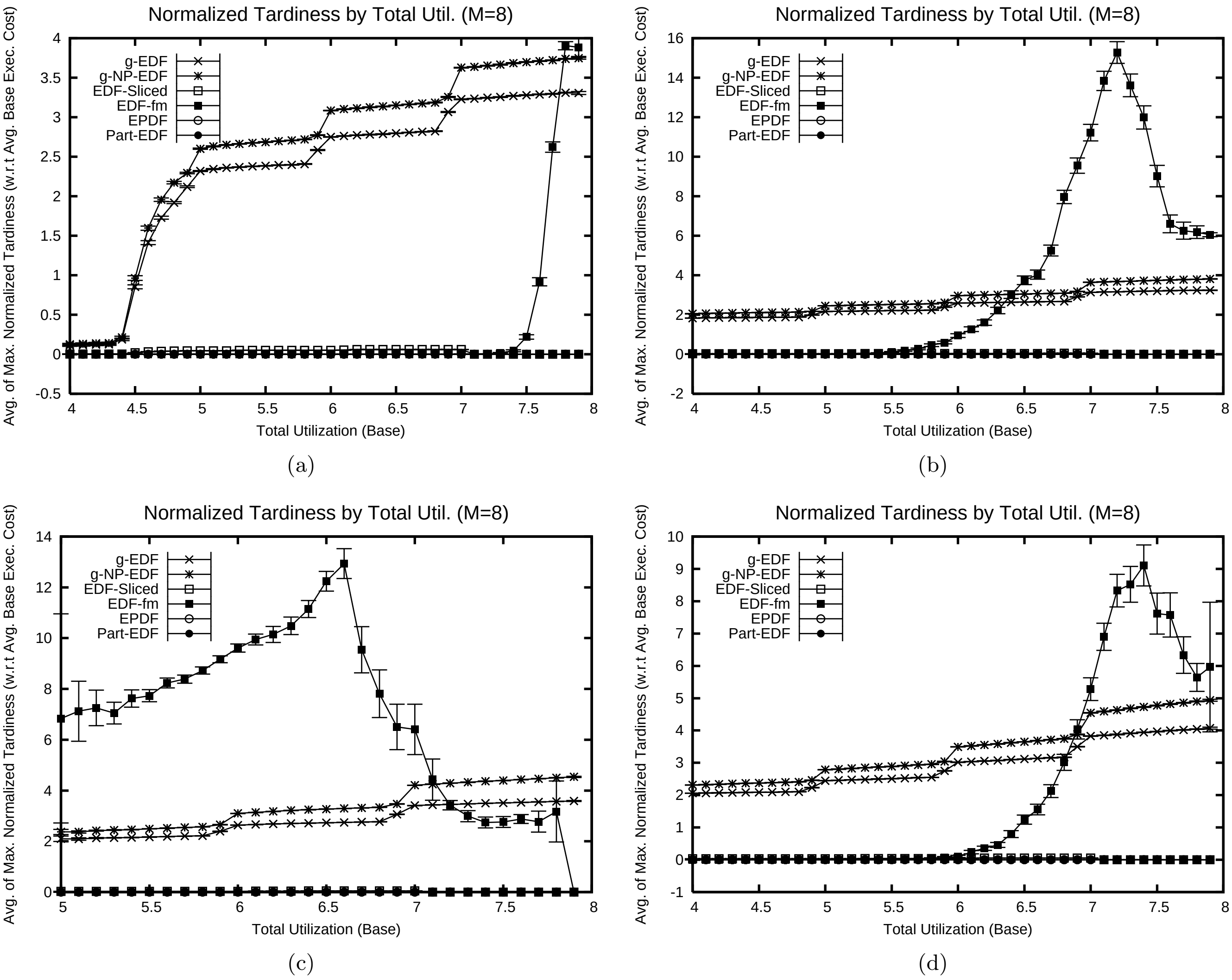


Figure 9.41: Comparison of tardiness bounds for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $232\mu s$, migration cost = $256\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

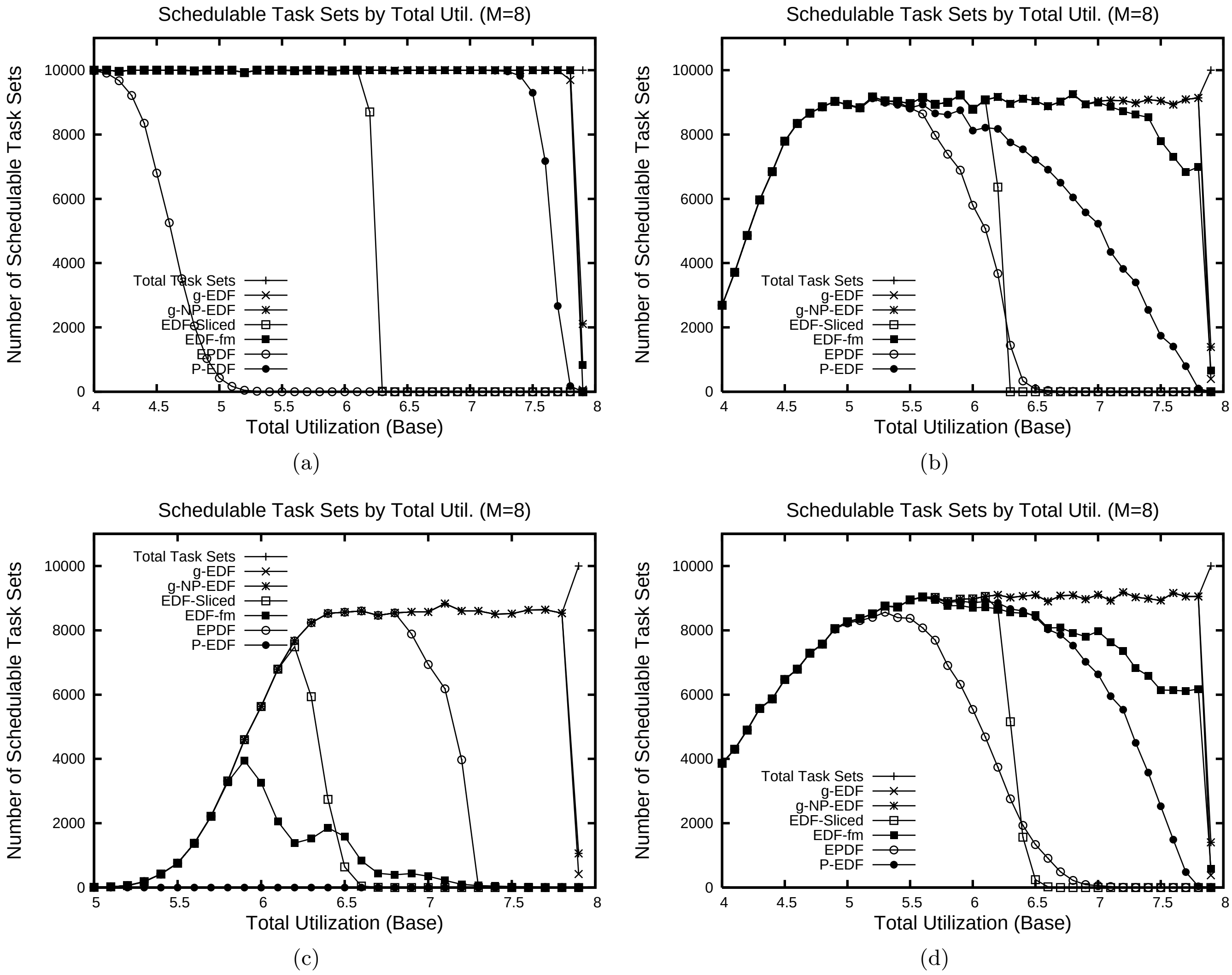


Figure 9.42: Schedulability comparison for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 128K (preemption cost = $464\mu s$, migration cost = $544\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

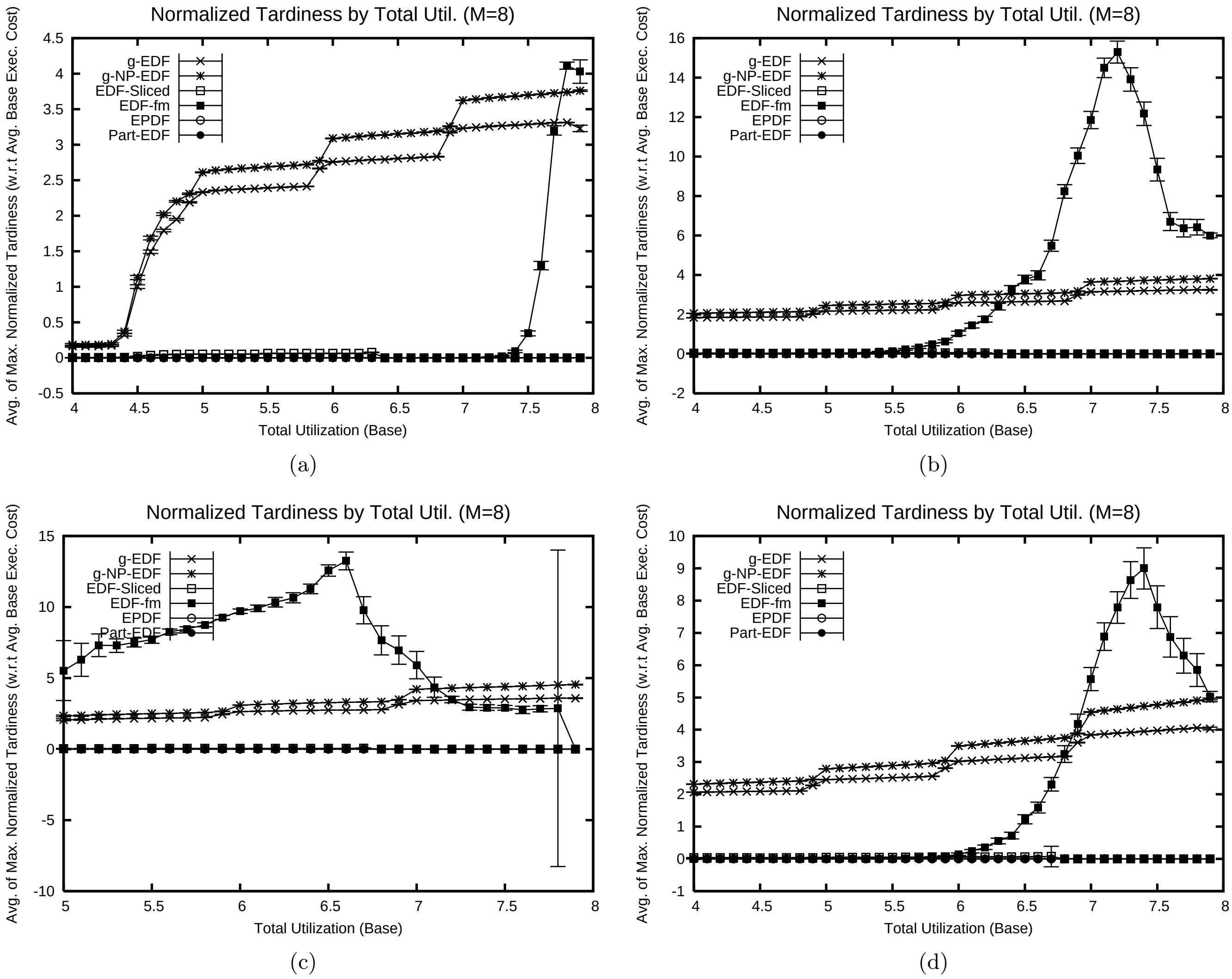
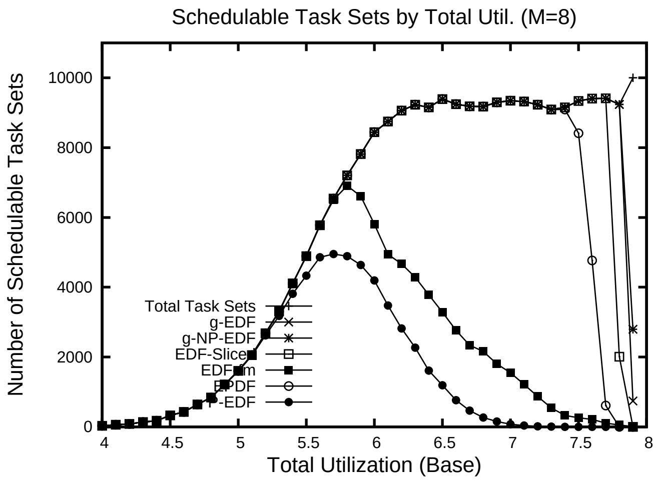
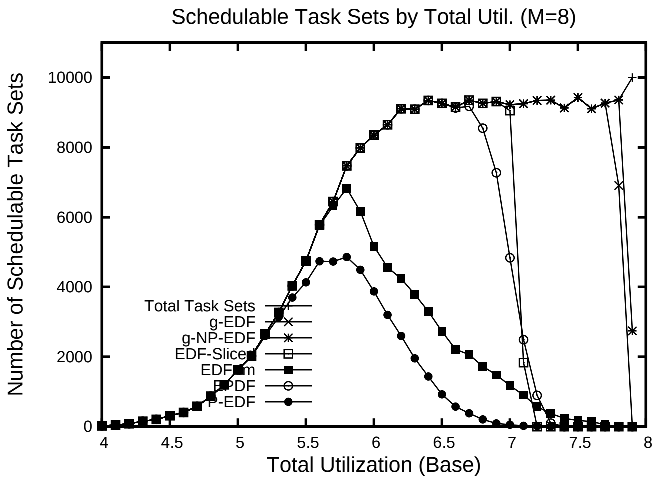


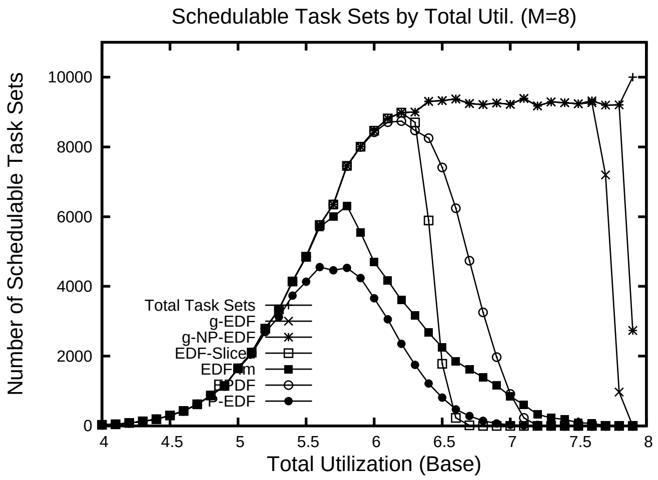
Figure 9.43: Comparison of tardiness bounds for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 256K (preemption cost = $464\mu s$, migration cost = $544\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.



(a)



(b)



(c)

Figure 9.44: Schedulability comparison with task utilizations distributed bimodally between the ranges $[0.1, 0.5)$ and $[0.5, 0.9)$ with probabilities 0.1 and 0.9, respectively, for $M = 8$, $Q = 1000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, and **(a)** WSS = 4K (preemption cost = $32\mu s$; migration cost = $32\mu s$), **(b)** WSS = 64K (preemption cost = $232\mu s$; migration cost = $256\mu s$), and **(c)** WSS = 128K (preemption cost = $464\mu s$; migration cost = $544\mu s$),

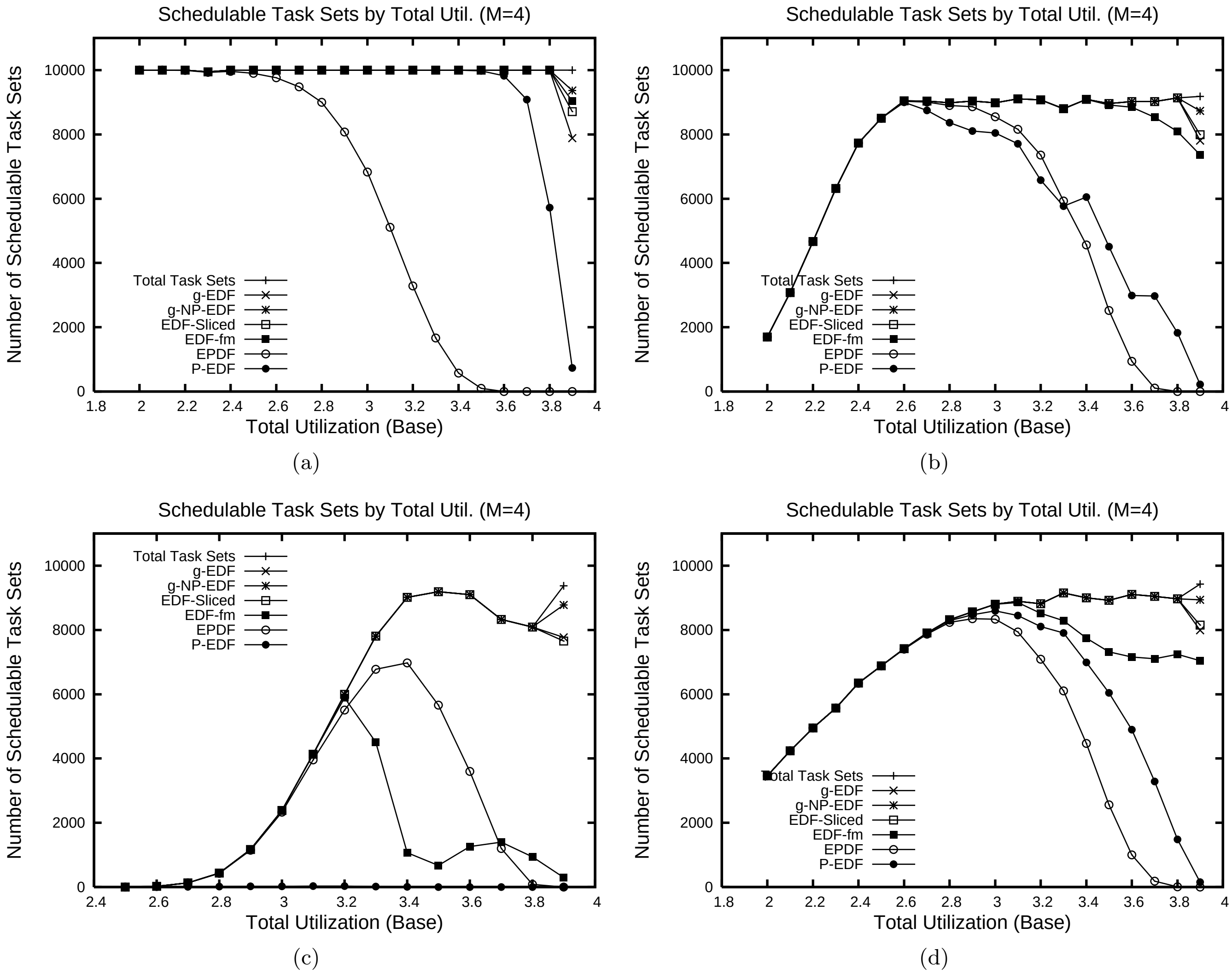


Figure 9.45: Schedulability comparison for $M = 4$, $Q = 5000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 4K (preemption cost = $16\mu s$, and migration cost = $16\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

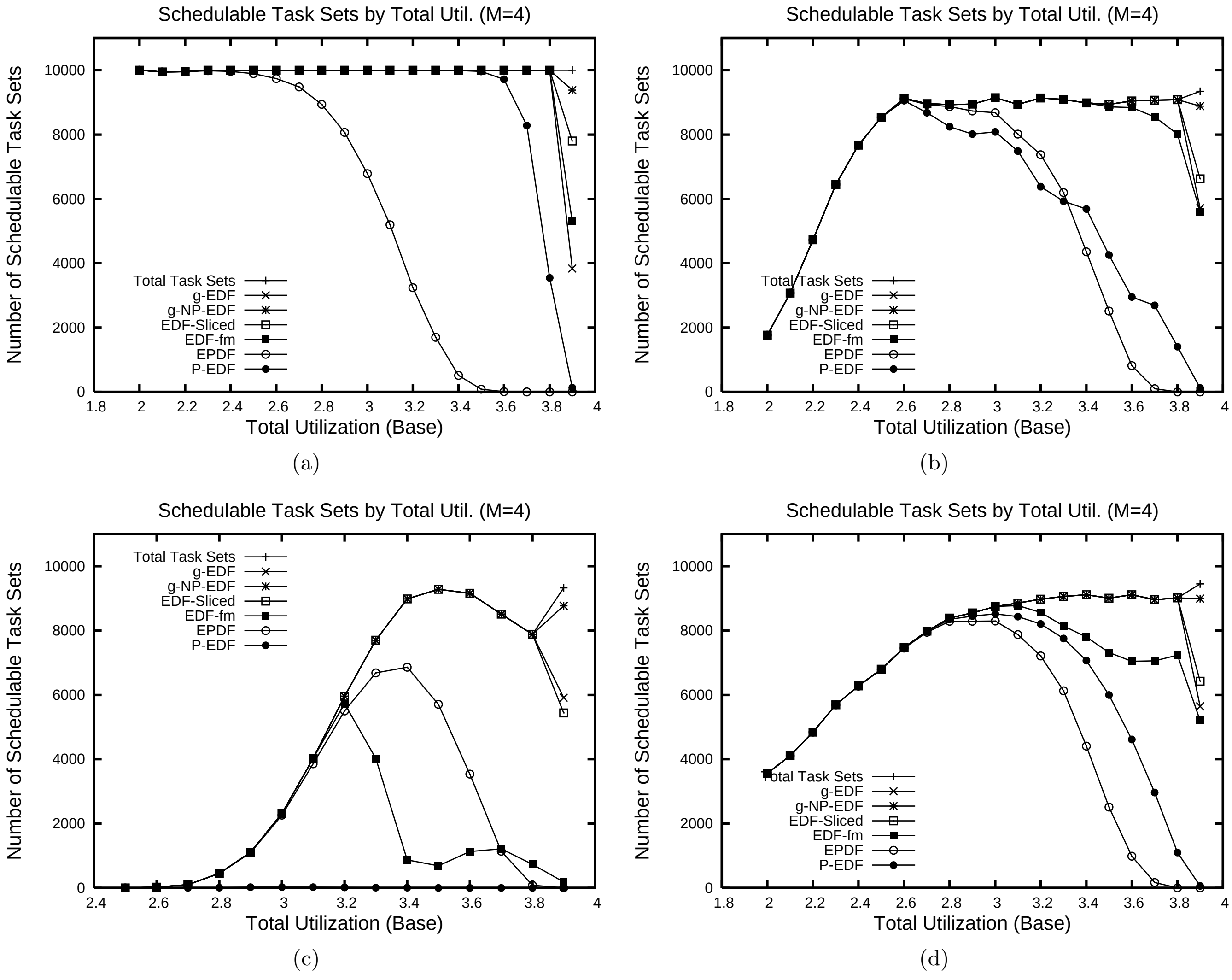


Figure 9.46: Schedulability comparison for $M = 4$, $Q = 5000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

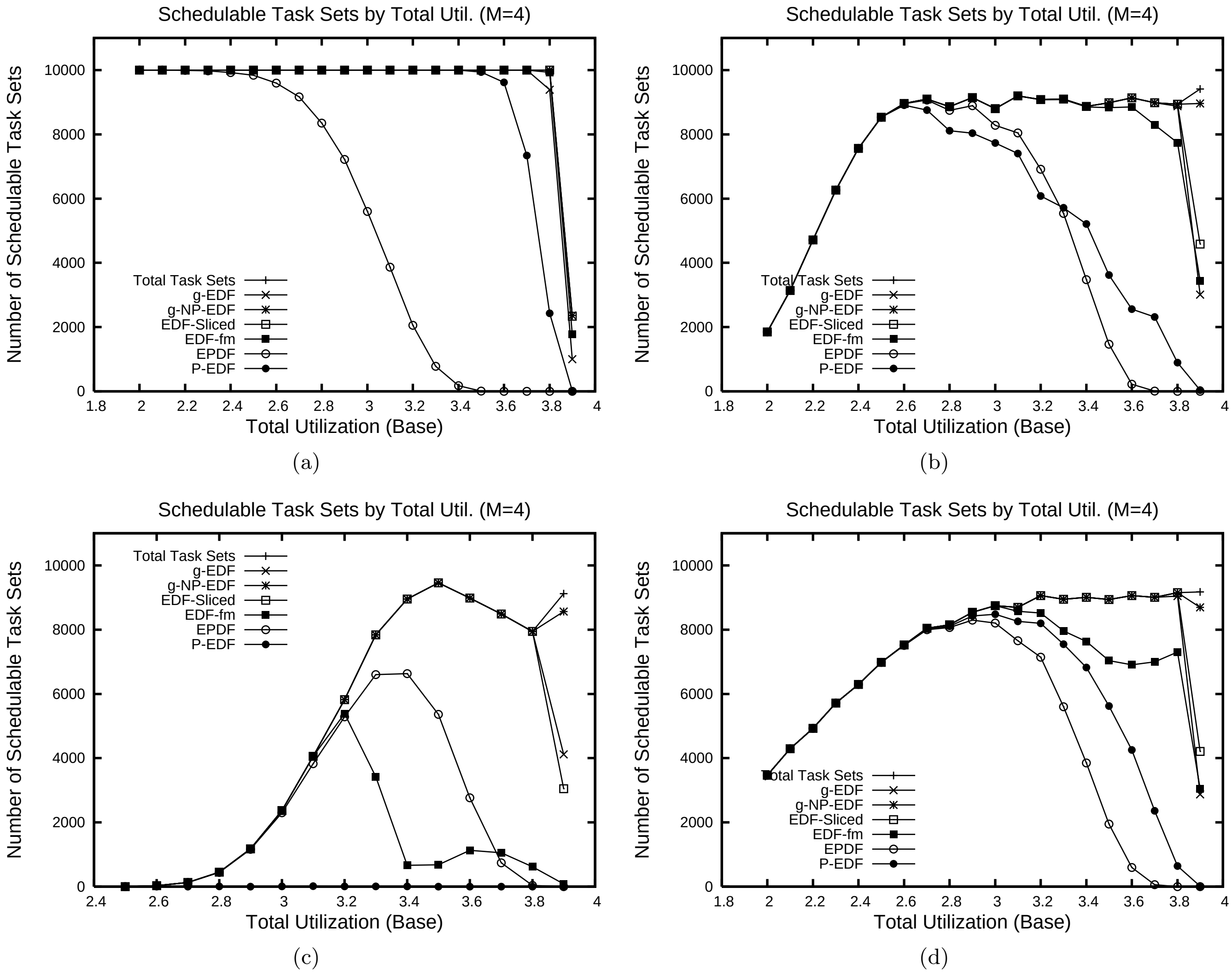


Figure 9.47: Schedulability comparison for $M = 4$, $Q = 5000\mu s$, $p_{\min} = 10ms$, $p_{\max} = 100ms$, WSS = 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

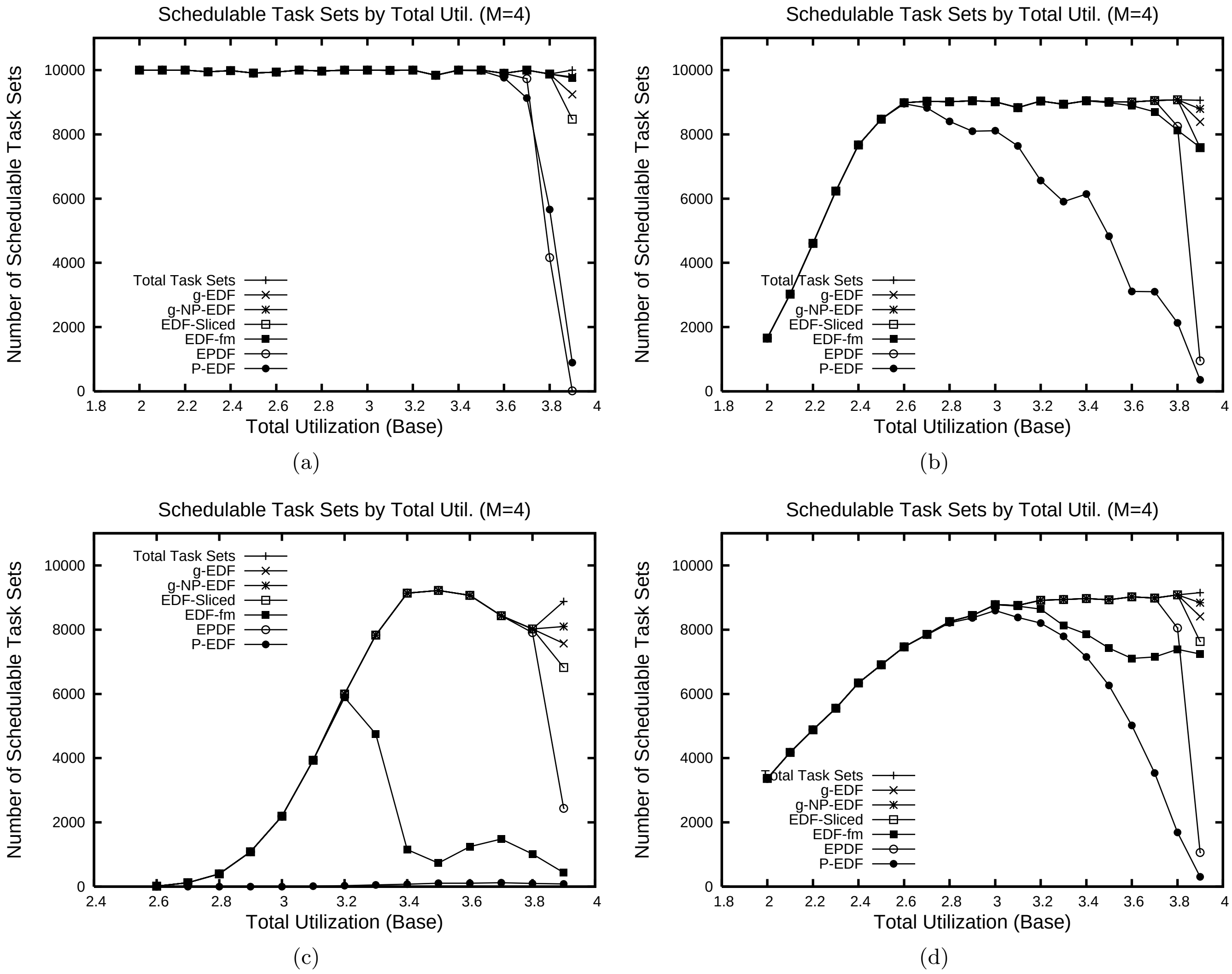


Figure 9.48: Schedulability comparison for $M = 4$, $Q = 5000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 4K (preemption cost = $16\mu s$, and migration cost = $16\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

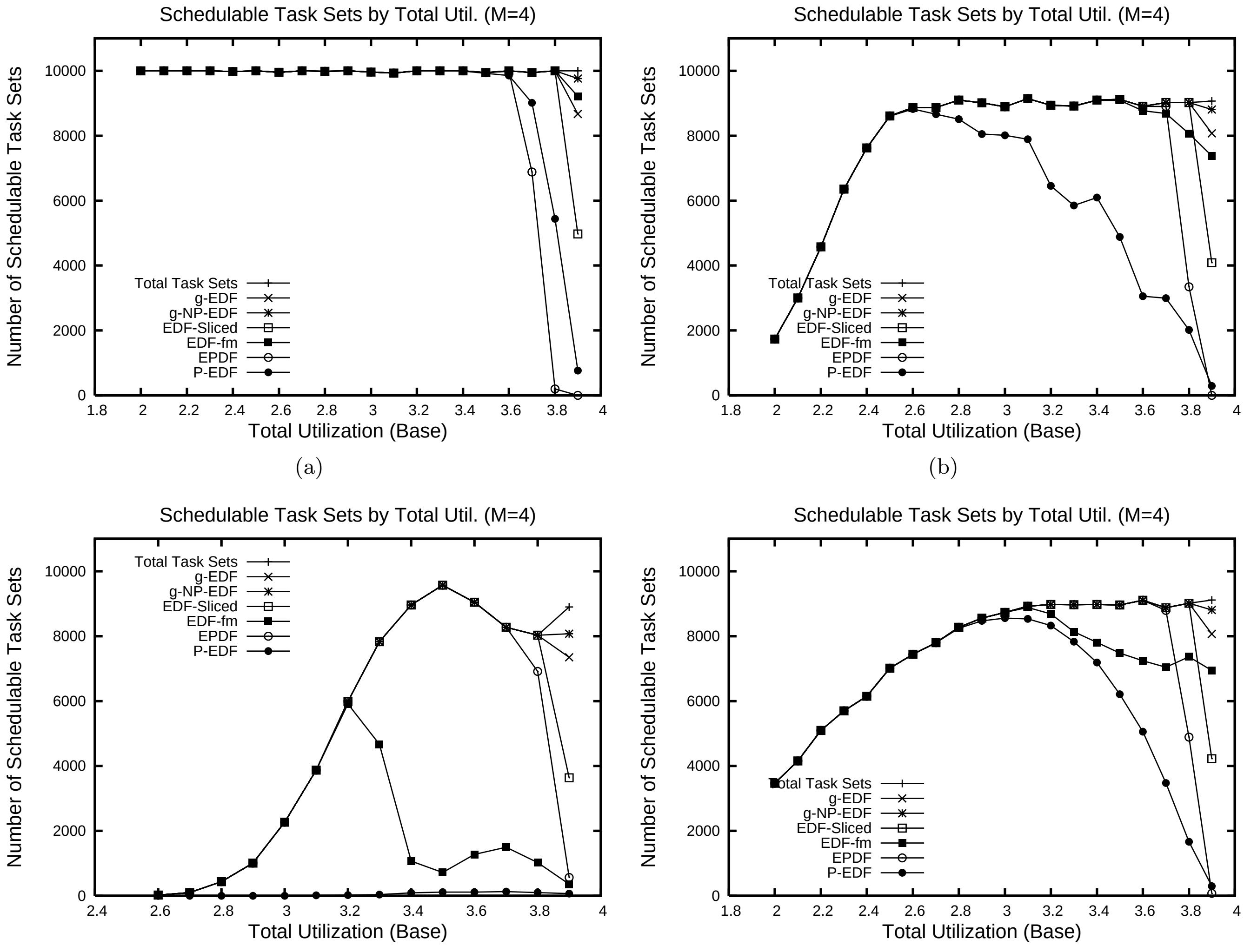
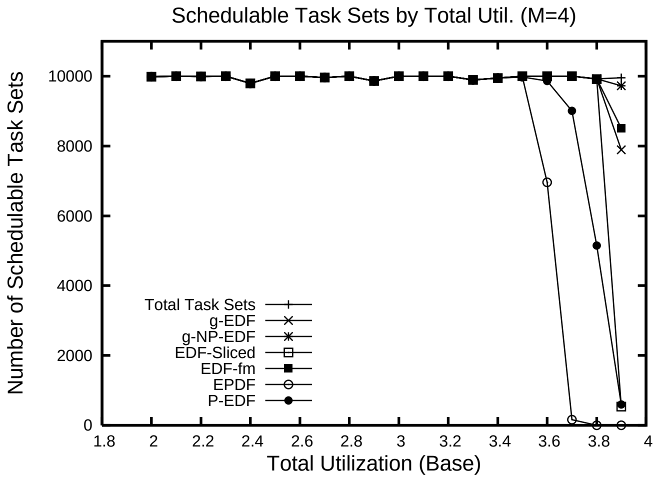
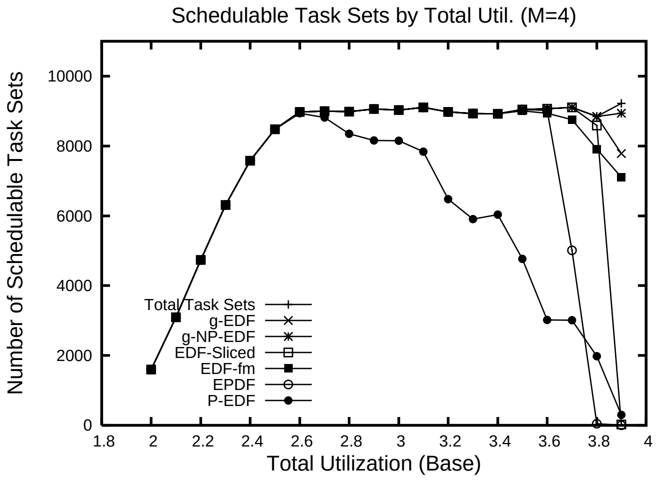


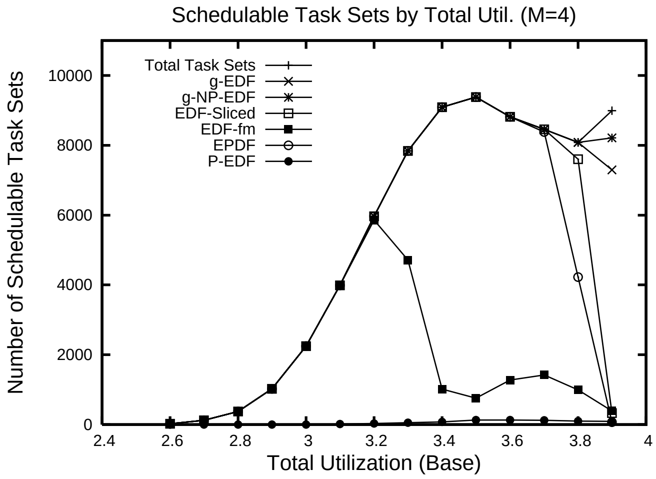
Figure 9.49: Schedulability comparison for $M = 4$, $Q = 5000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 64K (preemption cost = $116\mu s$, migration cost = $128\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.



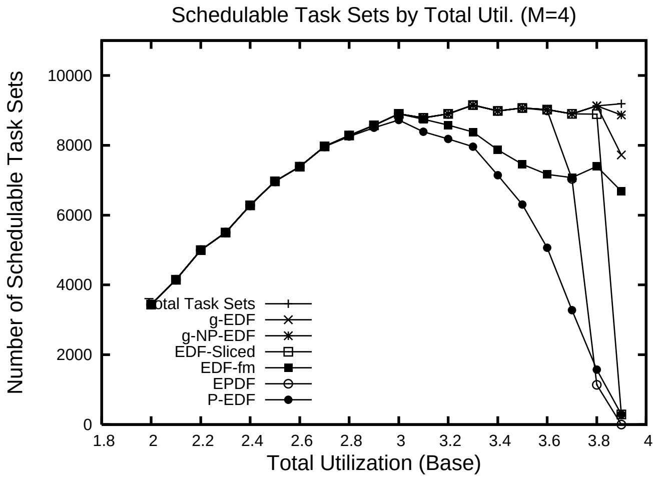
(a)



(b)



(c)



(d)

Figure 9.50: Schedulability comparison for $M = 4$, $Q = 5000\mu s$, $p_{\min} = 100ms$, $p_{\max} = 500ms$, WSS = 128K (preemption cost = $232\mu s$, migration cost = $272\mu s$), and **(a)** $[u_{\min}, u_{\max}) = [0.1, 0.5)$, **(b)** $[u_{\min}, u_{\max}) = [0.3, 0.7)$, **(c)** $[u_{\min}, u_{\max}) = [0.5, 0.9)$, and **(d)** $[u_{\min}, u_{\max}) = [0.1, 0.9)$.

9.5 Summary

In this chapter, we presented a simulation-based evaluation of p-EDF and the scheduling algorithms considered in this dissertation after accounting for some realistic overheads. Before presenting our evaluation, we described some significant practical overheads that can delay application tasks, showed how to account for those overheads under the different scheduling algorithms, and described the procedure we used to obtain some realistic values for the overheads. These real overheads were then used to drive an empirical study involving randomly-generated task sets to determine the percentage of task sets for which each algorithm can guarantee bounded tardiness and the maximum tardiness bound guaranteed on average. The results of our studies show that with respect to the task sets generated, the algorithms considered in this dissertation can guarantee bounded tardiness for a significant percentage of task sets that are not schedulable in a hard real-time sense. Furthermore, for each algorithm, conditions exist in which it may be the preferred choice.

Chapter 10

Conclusions and Future Work

In real-time systems research, enabling cost-effective implementations of soft real-time applications on multiprocessors is of growing importance. This is due to both an increase in the availability of affordable multiprocessor platforms, and the prevalence of applications with workloads necessitating multiprocessors and for which soft real-time guarantees are sufficient. One crucial aspect of a cost-effective system design lies in the careful and efficient allocation of system resources to application tasks — the task of the system scheduler.

However, real-time scheduling theory has not kept pace in providing needed tools and techniques for multiprocessor-based soft real-time systems. Specifically, known scheduling algorithms either are theoretically optimal but can incur significant overhead, which can reduce the amount of useful work accomplished, or are non-optimal and have been analyzed only in the context of hard real-time systems, and as such, can require that the workload be restricted to roughly 50% of the available processing capacity. Thus, state-of-the-art scheduling techniques can be overkill for soft real-time systems, which can tolerate occasional or bounded deadline misses, and hence, allow for a trade-off between timeliness and improved resource utilization. In this dissertation, we attempted to bridge this gap by analyzing some known algorithms in the context of guaranteeing bounded tardiness, designing a new algorithm specifically for soft real-time systems and analyzing it, and analyzing some relaxed variants of optimal Pfair algorithms. In this chapter, we conclude by summarizing the results presented in earlier chapters, and by enumerating and discussing some challenges that remain to be addressed.

10.1 Summary of Results

The thesis that this dissertation strived to support is that *processor utilization can be improved on multiprocessors while providing non-trivial soft real-time guarantees for different recurrent soft real-time applications, whose preemption and migration overheads can span different ranges and whose tolerances to tardiness are different, by designing new algorithms, simplifying optimal algorithms, and developing new validation tests.* In this section, we discuss results presented in support of this thesis.

In Chapter 4, we provided counterexamples that showed that improved resource utilization is not be possible for recurrent real-time task systems under partitioning algorithms or full-migration, static-priority algorithms even if bounded tardiness can be tolerated. Hence, in establishing our thesis, we focused on algorithms that use dynamic priorities (either restricted or unrestricted) and that allow some degree of migration. (Static-priority algorithms were not considered in general because as discussed in Section 1.4.2.2, within a migration class, such algorithms are strictly less powerful than and incur almost equal overhead as dynamic-priority algorithms.)

Analysis of preemptive and non-preemptive global EDF (g-EDF and g-NP-EDF). The elimination of partitioned and static-priority algorithms leaves the well-known EDF algorithm as an obvious first choice for consideration in soft real-time systems. Unlike optimal Pfair algorithms, EDF is a restricted dynamic-priority algorithm and incurs significantly less overhead. However, EDF suffers from the drawback that its schedulable utilization bound is low on multiprocessors. Specifically, on M processors, the worst-case schedulable utilization bound of g-EDF cannot exceed $(M+1)/2$ and the corresponding value is still lower for g-NP-EDF. Thus, g-EDF and g-NP-EDF offered significant scope in improving processor utilization for soft real-time systems if bounded tardiness that is reasonable can be guaranteed under them. Motivated by this prospect, in Chapter 4, we considered determining the extent to which deadlines can be missed by otherwise feasible task systems under g-EDF and g-NP-EDF. We showed that on M processors, g-EDF and g-NP-EDF can guarantee tardiness bounds of $\frac{\sum_{i=1}^{\Lambda} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda-1} \mu_i} + e_k$ and $\frac{\sum_{i=1}^{\Lambda+1} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda} \mu_i} + e_k$, respectively, for each task T_k of every task system τ with $U_{sum} \leq M$. (ϵ_i, μ_i, and Λ are defined in Section 4.2.) Though we do not believe the above bounds to be tight in general, as a special case, we derived a tardiness bound of $(e_{\max} + e_k)/2$ for two-processor systems under g-EDF, and provided a counterexample to show that this result is close to being tight. Further, as discussed in Chapter 4, the bounds are reasonable unless task utilizations and the number of processors are both high. Specifically, if task utilizations are at most 0.75 or the number of processors is at most eight, then the tardiness bounds guaranteed can be expected to be within tolerable limits. We expect most practical systems to adhere to these limits, and hence, the bounds derived should suffice for practical purposes.

Design and analysis of EDF-fm. Though migration and other overheads are lower under EDF than under optimal Pfair algorithms, migrations are still unrestricted. (Note that though g-NP-EDF eliminates job migrations by eliminating job preemptions, task migrations are still unrestricted.) This may be unappealing to some systems, especially to those with high inter- and intra-job migration costs and that cannot be partitioned. To address this issue, in Chapter 5, we proposed an algorithm called EDF-fm, which is based upon EDF, and treads a middle path, by restricting, but not eliminating, task migrations, and determined a tardiness bound that can be guaranteed under it. Specifically, under EDF-fm, the ability to migrate is required for at most $M-1$ tasks, and it is sufficient that every such task migrate between two processors and at job boundaries only. EDF-fm, like global EDF, can ensure bounded tardiness to a sporadic task system as long as the available processing capacity is not exceeded, but unlike global EDF, in general, only if the maximum task utilization, $u_{\max}$, is capped at 0.5. As $u_{\max}$ increases, the ability of EDF-fm to guarantee bounded tardiness decreases. However, in our simulation studies, we found that on eight processors, EDF-fm could guarantee bounded tardiness to approximately 60% of the generated task sets even when $u_{\max} = 1.0$.

Apart from deriving a tardiness bound that can be computed in $\mathcal{O}(N)$ time but is somewhat pessimistic, we also derived a pair of iterative formulas for computing near-exact bounds, but which may require exponential time. In order to lower the tardiness observed in practice, we proposed several heuristics for assigning to processors those tasks that do not migrate under EDF-fm, and through extensive simulations, evaluated the efficacy of these heuristics in lowering the tardiness bound that can be guaranteed. Finally, we presented a simulation-based evaluation of the accuracy of both the tardiness bounds derived under the heuristic identified to be the best.

Analysis of relaxed Pfair algorithms. Though g-EDF and g-NP-EDF can guarantee bounded tardiness to every feasible task system, their tardiness bounds may be beyond the tolerance limits of some systems. For such systems, algorithms based on Pfair scheduling may be better. Hence, in an attempt to enhance the applicability of Pfair scheduling for soft real-time systems, we considered relaxing some of its limiting restrictions.

First, we considered the earliest-pseudo-deadline-first (EPDF) Pfair scheduling algorithm, which avoids the use of tie-breaking rules, and hence, is more efficient than and preferable to optimal Pfair algorithms. EPDF is especially preferable in dynamic task systems with frequent changes in task composition or in systems implemented on less-powerful hardware platforms. However, EPDF is not optimal and deadlines can be missed under it if the number of processors exceeds two.

Before considering EPDF in the context of soft real-time systems, in Chapter 6, we determined a sufficient restriction on total system utilization for ensuring that no deadline will be missed under EPDF. Specifically, we showed that on M processors, EPDF can correctly schedule every task system τ with total utilization at most $\min(M, \frac{\lambda M(\lambda(1+W_{\max})-W_{\max})+1+W_{\max}}{\lambda^2(1+W_{\max})})$, where $W_{\max} = u_{\max}(\tau)$ and $\lambda = \max(2, \left\lceil \frac{1}{W_{\max}} \right\rceil)$. Apart from extending the applicability of EPDF for hard real-time systems, this result should also help in identifying the conditions for which the soft real-time scheduling results of EPDF are applicable.

In Chapter 7, we presented some soft real-time results for EPDF. We first showed that the prevailing conjecture that EPDF can guarantee a tardiness bound of one quantum to every feasible task system is false. Next, improving upon Srinivasan and Anderson's result that EPDF can guarantee a tardiness bound of q quanta, where $q \geq 1$, to every feasible task system with $W_{\max} \leq \frac{q}{q+1}$, we showed that a significantly more liberal restriction of $W_{\max} \leq \frac{q+2}{q+3}$ is sufficient for ensuring the same tardiness bound. Finally, we showed that if $W_{\max}$ exceeds $\frac{q+2}{q+3}$, then on M processors, a restriction of $\min(M, \frac{((q+1)W_{\max}+(q+2))M+((2q+1)W_{\max}+1)}{2(q+1)W_{\max}+2})$ on the total system utilization of a task system is sufficient to ensure that tardiness for that task system under EPDF is at most q.

In Chapter 8, we considered relaxing a restriction of Pfair algorithms that periods and execution costs be specified as integers. We showed that if periods are non-integral, then under any Pfair algorithm, tardiness is worsened by at most two quanta. For handling non-integer execution costs, we proposed a simple algorithm called EDF-sliced, under which, each job (except perhaps the final sub-job) of a task is divided into equal-size sub-jobs that are scheduled using an NP-EDF scheduling policy. Since NP-EDF does not require integral execution costs, loss due to rounding is eliminated; tardiness is lowered due to slicing albeit at the expense of incurring some migration overhead. Hence, for this algorithm, sub-job sizes provide a trade-off between loss due to migrations and tardiness.

Performance evaluation. Finally, in Chapter 9, we presented a simulation-based evaluation of p-EDF and the scheduling algorithms considered in this dissertation after accounting for some realistic overheads. Before presenting our evaluation, we described some significant practical overheads that can delay application tasks, showed how to account for those overheads under the different scheduling algorithms, and described the procedure we used to obtain some realistic measurements for the overheads. These real overheads were then used to drive an empirical study involving randomly-generated task sets to determine the percentage of task sets for which each algorithm can guarantee bounded tardiness and the tardiness bound guaranteed on average. The results of our studies showed that with respect to the task sets generated, bounded tardiness is guaranteed by the algorithms considered in this dissertation for a significant percentage of task sets that are not schedulable in a hard real-time sense. Furthermore, for each algorithm, conditions exist in which it may be the preferred choice.

10.2 Other Related Work

In this section, we briefly discuss some work by us related to the topic of this dissertation that is presented elsewhere but is not included here.

Guaranteeing flexible tardiness bounds. One limitation of the tardiness bounds derived in this dissertation for the various algorithms is that they are applicable to every task in the task system under consideration. In other words, any task can incur maximum tardiness. This may not be acceptable to applications that consist of hard and soft real-time tasks or those that consist of soft tasks with different tardiness tolerances. In [52], we considered the issue of supporting task systems with some tasks whose tolerances to tardiness are lower than that guaranteed under g-EDF. As a solution, we proposed a new scheduling policy, called EDF-hl, that is a variant of g-EDF, and show that under EDF-hl, any tardiness, including zero tardiness, can be ensured for a limited number of *privileged* tasks, and that bounded tardiness can be guaranteed to the remaining tasks if their utilizations are restricted. EDF-hl reduces to EDF in the absence of privileged tasks.

Desynchronized Pfair scheduling. In [50], we considered relaxing a restriction of Pfair scheduling that requires quanta to be synchronized on processors and tasks to be allocated processor time in units of fixed-sized quanta. We showed that if this requirement, which may lead to wasted processor time, is relaxed, then under an otherwise optimal Pfair scheduling algorithm, deadlines are missed by at most one quantum only, which is sufficient to provide soft real-time guarantees. This result can be shown to extend to most prior work on Pfair scheduling: In general, tardiness bounds guaranteed by non-optimal Pfair algorithms are worsened by at most one quantum only.

Real-Time scheduling on multicore platforms. In [11], we considered the architectural implications of multicore platforms for scheduling real-time tasks. Multicore architectures differ from conventional SMP architectures in that multiple processing units, referred to as processor cores, are housed in a single die. Most multicore designs additionally differ in the use of shared caches, which can be either on- or off-chip, for the various cores. While a shared cache has the potential to alleviate migration costs considerably, care must be taken to ensure that contention from tasks co-scheduled on the various cores does not lead to degraded performance. In prior work [57], Fedorova *et al.* observed that contention for shared caches can be a significant performance bottleneck in multicore systems and proposed a scheduling algorithm for improving throughput while executing non-real-time applications.

In [11], we considered the issue raised by Fedorova *et al.* in the context of scheduling real-time tasks. We proposed a hierarchical approach based on PD^2 for scheduling real-time tasks such that tasks that generate significant memory-to-L2 traffic are discouraged from being co-scheduled.

10.3 Future Work

We now discuss some of the challenges that remain in the area of multiprocessor-based soft real-time scheduling.

Tightening the tardiness bounds. Though this dissertation closes the problem of determining whether global EDF can guarantee bounded tardiness, the question remains as to how tight the bounds are. As mentioned earlier, our bounds can be high when the average task utilization is also high, say, above 0.8. While systems with such high-utilization tasks may be quite rare in practice, the problem of tightening the bounds is, nevertheless, theoretically important.

Support for other task models. This dissertation has determined some soft real-time guarantees that can be provided on multiprocessors when the workload is specified for the worst case, does not exceed the available processing capacity, and every job needs to execute in its entirety. Relaxing these assumptions is an obvious next step in the progression of the theory of soft real-time scheduling on multiprocessors. As discussed in Chapter 2, considerable progress has been made in this direction on uniprocessors. Intelligently skipping entire jobs as in the skippable task model and the (m, k)-firm model, or parts of them, as in the imprecise task model, can help in alleviating both long-term and transient short-term overloads for systems for which worst-case workload specifications are not pessimistic, *i.e.*, when the actual workload does not deviate significantly from what is specified. On the other hand, for applications whose workloads can vary widely over time, such as video-conferencing or animation games, reserving resources based on worst-case specifications can be extremely wasteful. For such applications, modeling task parameters using probabilistic distributions, and providing probabilistic guarantees may be better suited. Investigating the efficacy of alternative task models in the context of multiprocessors remains an open area of research.

Scheduling aperiodic tasks. In Chapter 2, we mentioned that one common approach for scheduling one-shot, aperiodic tasks that do not conform to any recurrent task model is by using server tasks. We also mentioned that, in general, a server task is either a periodic or a sporadic task and is assigned parameters based on the workload needs of the aperiodic tasks that it serves. However, assigning appropriate parameters for the server tasks is not simple (due to the variable nature of the aperiodic tasks), and on multiprocessors, is subject to additional complications than on uniprocessors [110].

While an ability to schedule aperiodic tasks independently in a stand-alone manner appears to be an attractive alternative, it is fraught with difficulties in ensuring that timing constraints will be met. This is because, for long-lived systems, offline analysis is simply infeasible due to the number of tasks, and online validation is not straightforward. Recently, Abdelzaher *et al.* considered scheduling aperiodic tasks in a stand-alone manner and developed utilization-based validation tests that can be applied at run-time for aperiodic tasks both on uniprocessors and multiprocessors [4, 3]. For this purpose, they defined a notion called *synthetic utilization* for aperiodic task sets that is based on the execution requirement and the relative deadline of each aperiodic task that has arrived, but not completed execution. They showed that, on uniprocessors, maintaining synthetic utilization below $\frac{1}{1+\sqrt{\frac{1}{2}}}$ (58.6%) is sufficient to ensure that no deadlines will be missed under the deadline-monotonic scheduling policy. On multiprocessors, they were able to establish a similar bound only for *liquid tasks*, that is, tasks whose execution requirements are infinitesimal in comparison to their relative deadlines.

On multiprocessors, one alternative to the approach of Abdelzaher *et al.* is to consider modeling aperiodic tasks using the extended sporadic task model proposed in Chapter 4, and use g-EDF as the scheduler. Intuition suggests that if bounded tardiness is tolerable, then the synthetic utilization can be up to 100% even for non-liquid tasks. It would be interesting to explore this idea further and evaluate it empirically.

Improving overhead accounting. In Chapter 9, we provided formulas for determining inflated WCETs of tasks that are sufficient to account for overheads incurred in practice. While the overhead costs that we determined are safe, it is unclear whether they are tight. Specifically, our assumption that under global algorithms, a preemption leads to a migration can be pessimistic. For example, if the number of tasks is small, say $M + 1$, it appears highly unlikely that each job preempts some job or each preemption leads to a migration. Furthermore, when jobs miss deadlines, there may be very few preemptions. Since migration costs are the most significant source of overhead, a better bound on the number of migrations has a high potential of lowering the amount of inflation needed and improving resource utilization. However, translating observations or intuitions into general results can be extremely difficult. To determine whether research in this direction is warranted, it may be helpful to determine, through simulations, the actual number of preemptions and migrations in schedules generated for random task sets.

Real-time operating systems. While most of the scheduling algorithms considered in this dissertation have been implemented in LITMUS$^{\text{RT}}$, currently support is provided only for scheduling independent, periodic tasks that are static. Adding support for sporadic tasks, and developing and implementing predictable and efficient techniques for supporting mutual exclusion and precedence constraints, and evaluating the different algorithms for sporadic, dynamic, and non-independent systems remain to be addressed. Furthermore, our evaluation was for small- to medium-sized conventional SMPs, and further work is needed for larger platforms, and other architectures, most notably, multicore platforms, as well. Finally, LITMUS$^{\text{RT}}$ currently uses tick scheduling for non-Pfair algorithms also. For such algorithms, it would be interesting to explore the viability of *tickless* implementations, wherein scheduling is entirely event driven.

Hierarchical scheduling for emerging architectures. Though existing multicore chips have only a modest number of cores, chips with 32 or more cores are expected within a decade [100]. As the number of cores increases, the complexity of cache configurations also increases as it may not be feasible for all cores to share a common cache at the lowest level. Hierarchical organizations, in which subsets of cores share common caches at the lowest level, with aggregation at higher levels is among the envisioned designs. In such designs, it may be preferable to limit task migrations to cores sharing lower level caches, and hence, hybrid scheduling algorithms, with elements of global and partitioned algorithms, may be needed. For instance, tasks can be partitioned among subsets of processor cores, and a global scheduling algorithm can be used for each subset. Note that, since the utilization of each task is at most one, the worst-case loss due to partitioning decreases as processor subset sizes increase. For example, if P denotes the number of subsets, n the number of cores per subset, and $u_{\max}$ the maximum utilization of any task, then a lower bound on the total utilization of a task set that cannot be partitioned is given by $P \cdot n - (P-1) \cdot u_{\max}$.[1] (Under traditional partitioning, $P = M$ and $n = 1$.) That is, the loss due to partitioning is at most $(P-1) \cdot u_{\max}$. When $n = 4$, the percentage loss due to partitioning is at most $(P-1) \cdot u_{\max}/4P$, and when $u_{\max}$ is slightly over 1/2, this loss reduces to $(P-1)/8P$, which is 12.5%. In contrast, the corresponding loss under traditional partitioning could be as high as 50%. Hence, if as expected, the cost of migrating among processors of the same subset is negligible, hybrid algorithms seem quite attractive. Further, it may even be possible to improve utilization by using techniques similar to those used in the design of EDF-fm. In any case, clearly, more research is needed on scheduling techniques for emerging architectures.

[1] It may be possible to determine a better bound than that provided here.

Appendix A

Remaining Proofs from Chapter 4

In this appendix, we provide the proofs omitted in Chapter 4 and discuss what eliminating the assumption in (4.1) entails.

A.1 Proof of Lemma 4.4

We first prove Lemma 4.4. (Lemma 4.4 is necessary to establish the bound as given in (4.43) for g-EDF. Otherwise, only a looser bound of $\frac{\sum_{i=1}^{\Lambda} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda} \mu_i} + e_k$ can be established. The difference is in the second term of the denominator. Also, without Lemma 4.4, the bound derived for two processors would not be close to being tight.)

Displacements. In proving Lemma 4.4, we consider removing one or more jobs from τ. Note that the resulting task system (composed of the remaining jobs) is still a valid concrete instantiation of τ^N. (We can simply assume either that jobs that are of the same task as any removed job and are released after that job are appropriately renumbered or that the actual execution costs of removed jobs are reduced to zero.) However, such job removals can cause one or more of the remaining jobs or job fragments to shift earlier in $\mathcal{S}$. A *job fragment* is defined with respect to a schedule, and is a portion of a job, which could potentially be a complete job, that executes continuously without preemption in some schedule under consideration. As described below, such shifts due to job removals can be reduced to a set of zero or more *disjoint* displacement chains comprised of job fragments of equal lengths. (If a job removal does not result in any change in the schedule for the remaining jobs, then the number of displacement chains is zero.) Each *displacement chain* Δ_i is denoted $(\Delta_{i,1}, \Delta_{i,2}, \ldots, \Delta_{i,n_i})$ and is a sequence of n_i equal-length and disjoint displacements. L denotes the length of every job fragment in every displacement. Each *displacement* $\Delta_{i,j}$ is a 4-tuple denoted $\langle J^{(i,j)}, t_{i,j}, J^{(i,j+1)}, t_{i,j+1}\rangle$, with the meaning that job fragment $J^{(i,j+1)}$ scheduled at $t_{i,j+1}$ displaces fragment $J^{(i,j)}$ scheduled at $t_{i,j}$. Here $J^{(i,j)}$ is the *displaced* job fragment and $J^{(i,j+1)}$, the *displacing* fragment. The displacements are disjoint in the sense that fragments $J^{(i,j)}$ and $J^{(i,j+1)}$ are disjoint. However, the

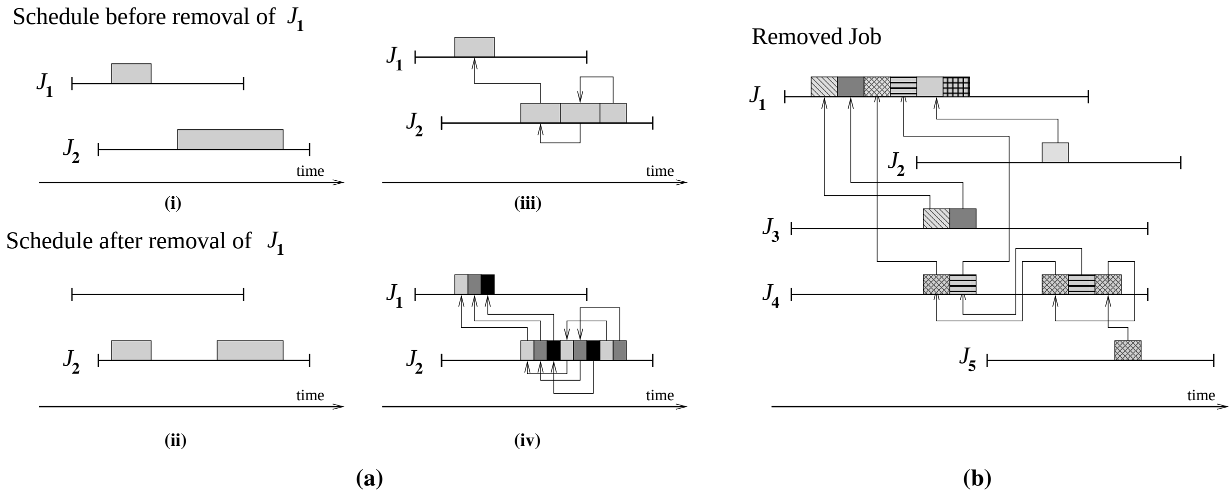


Figure A.1: Displacement chains resulting due to the removal of job J_1. **(a)** An example with simple shifts. **(a)-(ii)** Schedule for J_2 after the removal of J_1. **(a)-(iii)** A straightforward displacement chain, whose displacements are not of equal length. **(a)-(iv)** Decomposition of the displacement chain in **(a)-(iii)** into three chains of disjoint and equal-length displacements. **(b)** An example with complex shifts. In **(a)-(iv)** and **(b)**, job fragments that are parts of the same displacement chain are shaded identically and are linked using arrows. Arrows point from the displacing to the displaced fragment of each displacement.

time intervals in which the fragments are scheduled may overlap. Similarly, the displacement chains are disjoint in that no two chains can share a common job fragment or fragments that are overlapping. Finally, it should be noted that the displacing and displaced job fragments of a displacement can be part of the same job.

Informally, displacement chain Δ_i denotes, in sequence, job fragments whose schedules are altered due to the removal of the fragment at its head, namely, $J^{(i,1)}$. Hence, it is required that the displacing and displaced job fragments (*i.e.*, the second and first components) of two consecutive displacements in any chain be the same. Though it is not immediately obvious, it can be seen that by partitioning each removed job into as many fragments as necessary, all the shifts resulting from job removals can be reduced to a series of zero or more disjoint displacement chains of equal-length displacements. We will also make a reasonable assumption that the number of chains per removal is finite. Two examples are provided in Figure A.1. In the second example, the removed job J is partitioned into five fragments. Each fragment except the last is the head of a displacement chain of job fragments, which are linked.

Finally, a note on notation: a J with a two-tuple superscript as in $J^{(i,j)}$ denotes a fragment of an arbitrary job, and not a fragment of job J.

A few auxiliary lemmas are needed in establishing Lemma 4.4, but first we state the following simple claim. The claim follows because if jobs all fully preemptable, then jobs in Ψ are not impacted by those in $\overline{\Psi}$. Hence, a non-empty $\overline{\Psi}$ would contradict the assumption in (P2).

Claim A.1 *If* $b_{\max} = 0$ *then* $\overline{\Psi} = \emptyset$.

In the lemmas that follow, $b_{\max} = 0$ holds. Hence, EDF-P-NP reduces to g-EDF. To simplify description, in the proofs, we will assume that the scheduler is g-EDF.

The lemma below concerns displacements and is quite intuitive. It follows from the work-conserving nature of g-EDF and our assumption that deadline ties are resolved consistently.

Lemma A.1 *If* $b_{\max} = 0$*, then for every job removed from* τ*, the resulting sequence of shifts in* $\mathcal{S}$ *can be reduced to a set of zero or more displacement chains such that for each displacement* $\Delta_{i,j}$ *in each chain* Δ_i*,* $t_{i,j+1} > t_{i,j}$*, i.e., every displacement is to the left.*

We need one more lemma before proving Lemma 4.4.

Lemma A.2 *Let* $[t, t')$*, where* $t < t' \leq t_d$*, be a non-blocking, non-busy interval across which* LAG *for* Ψ *is continually increasing. If* $b_{\max} = 0$*, then there exists at least one job* J *such that* J*'s deadline is at or after* t' *and* J *completes executing at or before* t*.*

Proof: Contrary to the statement of the lemma, assume that there does not exist a job J as defined. First, we establish a part of the lemma in the following claim.

> **Claim A.2** *There exists at least one job* J *with deadline at or after* t' *that completes executing before* t'*.*
>
> **Proof:** Suppose J does not exist. Let $\hat{t}$ denote the latest of time t and the latest deadline of any job that completes executing before t'. By our assumption that a job like J does not exist, $\hat{t} \leq t'$ holds. Because $[\hat{t}, t')$ is a non-blocking, non-busy interval, there does not exist any instant in $[\hat{t}, t')$ at which a ready job from Ψ is waiting. Furthermore, because J does not exist, and by our choice of t', there is no instant in $[\hat{t}, t')$ at which there is an active job in Ψ that completed earlier, and hence, is not executing. Thus, there does not exist any instant in $[\hat{t}, t')$ where a task with an active or a pending job in Ψ is not executing. Hence, the total allocation at any instant u in the interval $[\hat{t}, t')$ to jobs in Ψ in $\mathcal{S}$ is $|\{\tau_i | \tau_{i,j} \text{is in } \Psi \text{ and is active or pending at } u\}|$, which, by (4.17) and (4.15), is at

least that those jobs receive in PS_τ. Therefore, by (4.19), LAG of Ψ at t' cannot exceed that at $\hat{t}$. This contradicts the fact that LAG for Ψ is continually increasing in $[t, t')$. ■

Note that J, as specified in the above claim, is not $\tau_{\ell,j}$. Otherwise, since J completes executing before its deadline, its tardiness is zero which contradicts (P1). Next, we show that, J completes executing at or before t. Assume to the contrary, and consider a concrete instantiation τ' of τ^N obtained by lowering the actual execution time of J such that J does not have to execute after t. Let $\mathcal{S}'$ be a schedule for τ' such that all ties among jobs are resolved the same way as in $\mathcal{S}$. Then in $\mathcal{S}'$, every job in τ' except J is scheduled exactly in the same intervals as in $\mathcal{S}$. This is because no job of Ψ is ready but waiting at any instant in $[t, t')$ in $\mathcal{S}$, and hence, the fact that another processor becomes available in $[t, t')$ due to J not executing in that interval would not cause additional jobs from Ψ to execute in $[t, t')$. Since $b_{\max} = 0$, by Claim A.1, $\overline{\Psi} = \emptyset$. Hence, no change to the schedule for jobs in Ψ due to the displacement of jobs in $\overline{\Psi}$ is possible. Thus, the completion time of every job in Ψ except J will be the same in both $\mathcal{S}$ and $\mathcal{S}'$. This contradicts (P3), since as explained above, J is not $\tau_{\ell,j}$. ■

We are finally ready to prove Lemma 4.4.

Lemma 4.4 *Let $[t, t')$, where $t < t' \leq t_d$, be a non-blocking, non-busy interval across which* LAG *for Ψ is continually increasing. If $b_{\max} = 0$, then there exists at least one job J and some time $\hat{t}$ such that $d(J) \geq t'$, $t \leq \hat{t} < t'$, and J executes continuously in $[\hat{t}, t')$.*

Proof: By Lemma A.2, there exists a job J' such that $d(J') \geq t'$ and J' completes executing by t. Because J' completes executing before its deadline, J' is not $\tau_{\ell,j}$. Otherwise, (P1) is contradicted. Our proof is by contradiction. Hence, assume (A) below holds.

(A) J as described in the statement of the lemma does not exist.

We show that if (A) holds, then (P2) does not hold, and therby derive a contradiction.

Consider a concrete instantiation τ' of τ^N obtained from τ by removing J'. (As explained above, J' is not $\tau_{\ell,j}$.) Let $\mathcal{S}'$ be a $\mathsf{g\text{-}EDF}$ schedule for τ' such that ties among jobs are resolved identically in $\mathcal{S}$ and $\mathcal{S}'$. To show that (P2) is contradicted, we show that if (A) holds, then the removal of J' does not result in any job fragment scheduled after t' in $\mathcal{S}$ (either partially or fully, *i.e.*, the fragment could have commenced execution before t') to shift left to before t' in

$\mathcal{S}'$. This would imply that no job fragments executing after t', and in particular those of $\tau_{\ell,j}$, shift earlier, and hence all jobs scheduled after t' have the same tardiness in both $\mathcal{S}$ and $\mathcal{S}'$.

Let the removal of J' result in a sequence of shifts for the remaining jobs and let these shifts be reduced to a set of equal-length, disjoint displacement chains. Our objective is to show that if some job fragment executing after t' gets displaced to the left, then (A) is contradicted. Hence, assume that some job fragment scheduled after t' undergoes a left shift. Because there is at least one displacement, the number of displacement chains is at least one, and by Lemma A.1, we have the following.

(D1) For every displacement $\Delta_{i,j}$ in every chain Δ_i, $t_{i,j+1} > t_{i,j}$.

Also, the fragment at the head of each displacement chain is that of job J', whose deadline is at or after t'. Therefore, by (D1), g-EDF prioritizes J' over the job of a fragment later in the chain. Hence, we have the following.

(D2) The deadline of the job of every fragment involved in every displacement is at least t'.

Let $\Delta_{i,k} = \langle J^{(i,k)}, t_{i,k}, J^{(i,k+1)}, t_{i,k+1} \rangle$ be a displacement such that $t_{i,k+1} + L \geq t'$, where L is the length of every fragment in the displacement chains under consideration, and $t_{i,k} < t'$. In other words, the displacement shifts the job fragment $J^{(i,k+1)}$ either partially or fully to the left of t'. (By (D1), $t_{i,k} < t_{i,k+1}$ holds.) If $t_{i,k+1} < t'$, then because $t_{i,k+1} + L \geq t'$, $J^{(i,k+1)}$ executes in $\mathcal{S}$ continuously in $[t_{i,k+1}, t')$, contradicting (A). On the other hand, if $t_{i,k+1} \geq t'$, then we argue as follows. Because $J^{(i,k+1)}$ is not executing before t' in $\mathcal{S}$ (implied by $t_{i,k+1} \geq t'$), the non-blocking, non-busy nature of $[t, t')$ and the fact that $J^{(i,k+1)}$ is executing before t in $\mathcal{S}'$ (implied by the displacement $\Delta_{i,k}$ and $t_{i,k} < t'$) imply that $J^{(i,k+1)}$ is not ready at t'^{-} in $\mathcal{S}$ because a prior job fragment, say, J^p, of the same task is executing there in $\mathcal{S}$. It also follows that $J^{(i,k+1)}$ is ready in $\mathcal{S}'$ before t' because J^p is displaced. By (D2), this implies that the deadline of the job of J^p is at or after t'. But then, because J^p is executing at t'^{-}, (A) is contradicted. Hence, if (A) holds, displacement $\Delta_{i,k}$ is not possible. That is, even if J' is removed, no job fragment can shift to the left of t', and so, the tardiness for every job completing execution after t' in $\mathcal{S}$, and in particular, for $\tau_{j,\ell}$, is the same in $\mathcal{S}'$, as well, which contradicts (P2). ■

A.2 Proofs of Lemmas 4.7 and 4.8

Lemma 4.7 *The following properties hold for sets* $\Gamma(k{+}c,\ell)$ *and* $\Pi(k{+}c,\ell)$ *with* $0 \le \ell \le k \le N$ *and* $0 \le c \le N-k$*, where* Γ *and* Π *are as defined in (4.3) and (4.4), respectively.*

(i) $\sum_{\tau_i \in \Gamma^{(k+c,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k+c,\ell)}} b_i \le \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \pi^{(k,\ell)}} b_i + \sum_{i=1}^{c} \beta_i$.

(ii) $\sum_{\tau_i \in \Gamma^{(k+c,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k+c,\ell)}} b_i \ge \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i$.

Proof: Let τ' denote the set of all tasks with ranks from $k+1$ to $k+c$ in a non-increasing ordering of the tasks by execution costs. Then,

$$|\tau'| = c \tag{A.1}$$

and $\Gamma^{(k+c,\ell)} \cup \Pi^{(k+c,\ell)} = \Gamma^{(k,\ell)} \cup \Pi^{(k,\ell)} \cup \tau'$ hold. Furthermore, $|\Gamma^{(k+c,\ell)}| = |\Gamma^{(k,\ell)}|$, $|\Pi^{(k+c,\ell)}| = |\Pi^{(k,\ell)}| + c$, and $\Pi^{(k,\ell)} \subseteq \Pi^{(k+c,\ell)}$ hold. (The final subset containment can be seen to hold as follows: if $\Gamma^{(k+1,\ell)} = \Gamma^{(k,\ell)}$, then $\Pi^{(k+1,\ell)}$ includes the task with rank $k+1$ in addition to every task in $\Pi^{(k,\ell)}$; on the other hand, if the task with rank $k+1$ is included in $\Gamma^{(k+1,\ell)}$, then some task from $\Gamma^{(k,\ell)}$ is added to $\Pi^{(k,\ell)}$ to form $\Pi^{(k+1,\ell)}$.) Define

$$A \subseteq \tau' \quad \wedge \quad B \subseteq \Gamma^{(k,\ell)} \tag{A.2}$$

such that

$$\Gamma^{(k+c,\ell)} = (\Gamma^{(k,\ell)} \setminus B) \cup A \tag{A.3}$$

holds. Note that, by (A.2), $A \cap B = \emptyset$, and by (A.2), (A.3), and (4.4), the following hold.

$$\begin{aligned} |A| &= |B| && \text{(A.4)} \\ \Pi^{(k+c,\ell)} &= \Pi^{(k,\ell)} \cup (\tau' \setminus A) \cup B && \text{(A.5)} \end{aligned}$$

Let $C = \tau' \setminus A$. Then,

$$\begin{aligned} &\sum_{\tau_i \in \Gamma^{(k+c,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k+c,\ell)}} b_i \\ &= \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in A} e_i - \sum_{\tau_i \in B} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i + \sum_{\tau_i \in C} b_i + \sum_{\tau_i \in B} b_i \qquad \text{(by (A.2), (A.3), and (A.5))} \end{aligned}$$

$$\leq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i + \sum_{\tau_i \in C} b_i + \sum_{\tau_i \in B} b_i$$ (by (A.2) and the definitions of τ' and $\Gamma^{(k,\ell)}$, the execution costs of tasks in A are at most those of tasks in B, and by (A.4), $|A| = |B|$)

$$\leq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i + \sum_{i=1}^{c} \beta_i$$ (by (A.4) and $C = \tau' \setminus A$, $|B| + |C| = |\tau'|$; by (A.1), $|\tau'| = c$; by their definitions, B and C are disjoint).

The above establishes Part (i). Part (ii) can be shown to hold as follows. Because tasks in A are in $\Gamma^{(k+c,\ell)}$, whereas those in B are in $\Pi^{(k+c,\ell)}$, by (4.3) and (4.4), we have

$$(\forall \tau_i, \tau_j : \tau_i \in A, \tau_j \in B :: e_i - b_i \geq e_j - b_j). \tag{A.6}$$

Finally,

$$\begin{aligned}
&\sum_{\tau_i \in \Gamma^{(k+c,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k+c,\ell)}} b_i \\
&= \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in A} e_i - \sum_{\tau_i \in B} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i + \sum_{\tau_i \in C} b_i + \sum_{\tau_i \in B} b_i \quad \text{(by (A.2), (A.3), and (A.5)} \\
&\geq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i + \sum_{\tau_i \in A} b_i + \sum_{\tau_i \in C} b_i \quad \text{(by (A.6), since } |A| = |B| \text{ by (A.4))} \\
&\geq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i,
\end{aligned}$$

establishing Part (ii). ∎

We prove two auxiliary lemmas before proving Lemma 4.8.

Lemma A.3 $\sum_{\tau_i \in \Gamma^{(k,\ell+c)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell+c)}} b_i \geq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i$, *where* $0 \leq \ell \leq k \leq N$ *and* $0 \leq c \leq k - \ell$.

Proof: The proof is by induction on c. It is easy to see that the lemma holds trivially when $c = 0$, which forms the base case. For the induction hypothesis, assume that the lemma holds for all c, where $0 \leq c \leq c'$, where $c' < k - \ell$. For the induction step, we show that $\sum_{\tau_i \in \Gamma^{(k,\ell+c'+1)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell+c'+1)}} b_i \geq \sum_{\tau_i \in \Gamma^{(k,\ell+c')}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell+c')}} b_i$, which, by the induction hypothesis establishes the lemma. By (4.2) and (4.3), $\Gamma^{(k,\ell+c'+1)}$ includes one more task that is in $\Gamma^{(k)}$ in addition to every task in $\Gamma^{(k,\ell+c')}$. By (4.4), this additional task, say τ_j, is from $\Pi^{(k,\ell+c')}$. Also, $\Pi^{(k,\ell+c'+1)}$ contains every task in $\Pi^{(k,\ell+c')}$ except τ_j. Therefore,

$$\sum_{\tau_i \in \Gamma^{(k,\ell+c'+1)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell+c'+1)}} b_i = \sum_{\tau_i \in \Gamma^{(k,\ell+c')}} e_i + e_j + \sum_{\tau_i \in \Pi^{(k,\ell+c')}} b_i - b_j$$

$$\geq \sum_{\tau_i \in \Gamma^{(k,\ell+c')}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell+c')}} b_i \qquad \text{(because } e_j \geq b_j\text{).}$$

■

Lemma A.4 *Let α and β be any two* ***disjoint*** *subsets of tasks in τ such that $|\alpha| = k_2$ and $|\alpha| + |\beta| = k_1$. Then, $\sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta} b_i \leq \sum_{\tau_i \in \Gamma^{(k_1,k_2)}} e_i + \sum_{\tau_i \in \Pi^{(k_1,k_2)}} b_i$.*

Proof: To prove this lemma, we need to show that there do not exist disjoint subsets α and β of τ with cardinalities k_2 and $k_1 - k_2$, respectively, such that the sum of the execution costs of tasks in α and the non-preemptive section costs of tasks in β is greater than the sum of the execution costs of tasks in $\Gamma^{(k_1,k_2)}$ and the non-preemptive costs of tasks in $\Pi^{(k_1,k_2)}$.

Let $\tau' = \tau \setminus (\Gamma^{(k_1,k_2)} \cup \Pi^{(k_1,k_2)})$. Then, by (4.5), $\tau' = \tau \setminus \Gamma^{(k_1)}$. By (4.3), $\Gamma^{(k_1,k_2)} \subseteq \Gamma^{(k_1)}$ holds. Therefore, by the definition in (4.2), no task in τ' has a higher execution cost than a task in $\Gamma^{(k_1,k_2)}$. Similarly, by (4.4), $\Pi^{(k_1,k_2)} \subseteq \Gamma^{(k_1)}$ holds, and, hence, by (4.2) and (4.1), no task in τ' has a higher non-preemptive section cost than a task in $\Pi^{(k_1,k_2)}$. Therefore, replacing a task in $\Gamma^{(k_1,k_2)}$ or $\Pi^{(k_1,k_2)}$ by a task in τ' will not lead to a pair of subsets α and β such that the lemma is contradicted. Hence, we are left with showing that there do not exist subsets α and β, and tasks τ_i and τ_j, such that $\alpha \cup \beta = \Gamma^{(k_1,k_2)} \cup \Pi^{(k_1,k_2)}$, $\tau_i \in \Gamma^{(k_1,k_2)}$, and $\tau_j \in \Pi^{(k_1,k_2)}$, and $\alpha = (\Gamma^{(k_1,k_2)} \setminus \tau_i) \cup \{\tau_j\}$ and $\beta = (\Pi^{(k_1,k_2)} \setminus \tau_j) \cup \{\tau_i\}$, for which $\sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta} b_i > \sum_{\tau_i \in \Gamma^{(k_1,k_2)}} e_i + \sum_{\tau_i \in \Pi^{(k_1,k_2)}} b_i$. (That is, we are left with showing that α and β derived from $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$ by interchanging the set memberships of tasks τ_i and τ_j cannot contradict the lemma.) Since τ_i and τ_j are as defined, it suffices to show that $e_i + b_j \geq e_j + b_i$. For this, note that by the definitions in (4.3) and (4.4), $e_i - b_i \geq e_j - b_j$ holds, which implies $e_i + b_j \geq e_j + b_i$. ■

Lemma 4.8 *Let α and β be any two* ***disjoint*** *subsets of tasks in τ such that $|\alpha| \leq \ell$ and $|\alpha| + |\beta| \leq k$, where $0 \leq \ell \leq k \leq N$. Then, $\sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta} b_i \leq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i$.*

Proof: By Lemma A.4, $\sum_{\tau_i \in \alpha} e_i + \sum_{\tau_i \in \beta} b_i \leq \sum_{\tau_i \in \Gamma^{(|\alpha|+|\beta|,|\alpha|)}} e_i + \sum_{\tau_i \in \Pi^{(|\alpha|+|\beta|,|\alpha|)}} b_i$ holds. By part (ii) of Lemma 4.7, $\sum_{\tau_i \in \Gamma^{(|\alpha|+|\beta|,|\alpha|)}} e_i + \sum_{\tau_i \in \Pi^{(|\alpha|+|\beta|,|\alpha|)}} b_i \leq \sum_{\tau_i \in \Gamma^{(k,|\alpha|)}} e_i + \sum_{\tau_i \in \Pi^{(k,|\alpha|)}} b_i$ holds. Lastly, by Lemma A.3, $\sum_{\tau_i \in \Gamma^{(k,|\alpha|)}} e_i + \sum_{\tau_i \in \Pi^{(k,|\alpha|)}} b_i \leq \sum_{\tau_i \in \Gamma^{(k,\ell)}} e_i + \sum_{\tau_i \in \Pi^{(k,\ell)}} b_i$ holds, establishing Lemma 4.8. ■

A.3 Eliminating the Assumption in (4.1)

The assumption in (4.1) is used in Lemmas 4.9 and 4.10 in simplifying expressions involving execution costs and non-preemptive section costs of subsets of tasks in τ. If (4.1) does not hold and if subsets Γ and Π are as defined in (4.3) and (4.4), respectively, then such simplifications, and hence, the bound in Theorem (4.1), cannot be shown to hold. In this section, we show how to define $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$ so that the tardiness bounds derived in Section 4.3 still hold (when the assumption in (4.1) is removed). Recall that $k_2 \leq k_1 \leq N$. (As mentioned in Section 4.2, the bound in Theorem 4.1 will be only slightly higher if $\Gamma^{(k_1,k_2)}$ is simply defined as a subset of k_2 tasks of τ with the highest execution costs, and $\Pi^{(k_1,k_2)}$ as a subset of $k_1 - k_2$ tasks with the highest non-preemptive costs, where the two subsets are not necessarily disjoint. However, if the overall bound is tightened further in the future, then the difference could be significant.)

As mentioned in Section 4.2, if (4.1) does not hold, then tasks for the subsets have to be chosen using an algorithm that is difficult to express as a closed-form expression. One such algorithm is given in Figure A.2.

Recall from Section 4.2 that tasks for $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$ are to be chosen such that there do not exist disjoint subsets τ^1 and τ^2 of τ with k_2 and $k_1 - k_2$ tasks, respectively, such that the sum of the execution costs of tasks in $\Gamma^{(k_1,k_2)}$ and the non-preemptive section costs of tasks in $\Pi^{(k_1,k_2)}$ is less than the sum of the execution costs of tasks in τ^1 and the non-preemptive costs of tasks in τ^2, *i.e.*, Lemma A.4 holds.

For conciseness, let $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$ be denoted Γ and Π respectively. Based on how tasks are chosen in lines 1 and 6 of the task selection algorithm Choose-Γ-Π in Figure A.2, we have the following.

$$|\Gamma_0| = |\Gamma_1| = |\Gamma| = k_2 \tag{A.7}$$

$$|\Pi_0| = |\Pi_1| = |\Pi| = k_1 - k_2 \tag{A.8}$$

$$\Gamma_1 \cup \Pi_1 = \Gamma_0 \cup \Pi_0 \qquad \text{(by lines 3 and 4)} \tag{A.9}$$

$$(\forall \tau_i \in \Gamma_1, \tau_j \in \Pi_1 :: e_i - b_i \leq e_j - b_j) \qquad \text{(by lines 3 and 4)} \tag{A.10}$$

$$\Gamma \cap \Pi_1 = \Gamma \cap \Pi = \emptyset \qquad \text{(by lines 5 and 6)} \tag{A.11}$$

$$\Gamma_1 \cap \Pi_1 = \Gamma_0 \cap \Pi_0 = \emptyset \qquad \text{(by lines 1 – 4)} \tag{A.12}$$

Algorithm Choose-Γ-Π (k_1, k_2) $\quad\quad$ $\triangleright$ $k_2 \leq k_1 \leq N$

$\triangleright$ Ties in choosing tasks for the following subsets are resolved based on task indices

1 $\Gamma_0 :=$ subset of k_2 tasks of τ with highest execution costs;
2 $\Pi_0 :=$ subset of $k_1 - k_2$ tasks of $\tau \setminus \Gamma_0$ with highest non-preemptive section costs;
3 $\Gamma_1 :=$ subset of k_2 tasks τ_i of $\Gamma_0 \cup \Pi_0$ with highest values for $e_i - b_i$;
4 $\Pi_1 := (\Gamma_0 \cup \Pi_0) \setminus \Gamma_1$;
5 $\Gamma := \Gamma^{(k_1,k_2)} := k_2$ tasks from $\tau \setminus \Pi_1$ with highest execution costs;
6 $\Pi := \Pi^{(k_1,k_2)} := \Pi_1$

Figure A.2: An algorithm for choosing tasks for $\Gamma^{(k_1,k_2)}$ and $\Pi^{(k_1,k_2)}$ when (4.1) does not hold.

By the assignments in lines 6 and 4,

$$\Pi \subseteq \Gamma_0 \cup \Pi_0 \tag{A.13}$$

holds.

Suppose $\Gamma \neq \Gamma_1$, and suppose τ_i is not in Γ but is in Γ_1. Then, because the cardinalities of the two sets are equal, there exists a task τ_j that is not in Γ_1 but is in Γ. Also, because $\Gamma \cap \Pi_1 = \emptyset$ and τ_j is in Γ, it follows that τ_j is not in Π_1. Further, line 1 implies that the execution cost of any task in Γ_0 at least as much as that of a task not in Γ_0. Therefore, because τ_i is in Γ_1, but not in Γ, $|\Gamma_0| = |\Gamma|$ holds, and tasks are chosen in line 5 in non-increasing order of execution costs, τ_i cannot be in Γ_0. Hence by (A.9), τ_i is in Π_0. That is, the following holds.

$$\tau_i \notin \Gamma \ \wedge \ \tau_i \in \Gamma_1 \ \Rightarrow \ \tau_i \notin \Gamma_0 \ \wedge \ \tau_i \in \Pi_0 \ \wedge \ (\exists \tau_j \mid \tau_j \in (\tau \setminus (\Gamma_1 \cup \Pi_1)) \wedge \ \tau_j \in \Gamma) \tag{A.14}$$

Similarly, we have the following.

$$\tau_i \in \Gamma \ \wedge \ \tau_i \notin \Gamma_1 \ \Rightarrow \ (\exists \tau_j \notin \Gamma \wedge \ \tau_j \in \Gamma_1) \tag{A.15}$$

Let τ' denote the subset of all tasks of τ that are neither in Γ nor in Π, *i.e.*,

$$\tau' = \tau \setminus (\Gamma \cup \Pi). \tag{A.16}$$

To show that Algorithm Choose-Γ-Π satisfies the needed property (that the sum of the execution costs of tasks in Γ and the non-preemptive section costs of tasks in Π is at least as much as that of the execution costs of tasks in τ^1 and the non-preemptive section costs of tasks in τ^2, where τ^1 and τ^2 are arbitrary, disjoint subsets of τ such that $|\tau^1| = k_2$ and $|\tau^2| = k_1 - k_2$), we need to show that the following hold.

$$(\forall \tau_i \in \Gamma, \tau_j \in \Pi :: (e_i + b_j) \leq (e_j + b_i)) \tag{A.17}$$

$$(\forall \tau_k \in \tau', \tau_i \in \Gamma :: e_k \leq e_i) \tag{A.18}$$

$$(\forall \tau_k \in \tau', \tau_j \in \Pi :: b_k \leq b_j) \tag{A.19}$$

Proof of (A.17). (A.17) simply states that interchanging the memberships of two tasks τ_i and τ_j in Γ and Π, respectively, is not needed. By (A.9), $\Gamma_1 \cup \Pi_1 = \Gamma_0 \cup \Pi_0$ and by the assignment in line 6, $\Pi = \Pi_1$. Hence, (A.17) holds for every τ_i in Γ that is also in Γ_1 by the way tasks in Γ_1 and Π_1 are selected in lines 3 and 4, respectively. Therefore, for the rest of this proof assume τ_i is not in Γ_1. Then, because τ_i is in Γ, by (A.15), there exists a τ_k such that τ_k is in Γ_1 but not in Γ. By line 1, the execution cost of any task in Γ_0 is at least the execution cost of a task not in Γ_0. Hence, since tasks for Γ are chosen in non-increasing order of execution costs, and the cardinalities of Γ and Γ_0 are equal, $\tau_k \notin \Gamma_0$. Therefore, since τ_k is in Γ_1, (A.9) implies that τ_k is in Π_0. Since τ_i is not in Π (because it is in Γ), by line 6, τ_i is not in Π_1, and hence, not in $\Gamma_1 \cup \Pi_1$ (because as discussed above, it is not in Γ_1), and by (A.9), not in $\Gamma_0 \cup \Pi_0$. So, because τ_k is in Π_0,

$$b_k \geq b_i \tag{A.20}$$

holds. Also, since τ_k is in Γ_1 whereas τ_j is in Π_1, the way tasks are selected in lines 3 and 4 implies that

$$e_k - b_k \geq e_j - b_j. \tag{A.21}$$

Finally, since τ_k is not in Γ, whereas τ_i is, $e_i \geq e_k$ holds. Therefore, by (A.20) and (A.21), $e_i - b_i \geq e_j - b_j$ holds from which (A.17) follows. ■

Proof of (A.18). Is immediate from how tasks are chosen for Γ in line 5. ■

Proof of (A.19). If $\tau_j \in \Pi_0$, then (A.19) follows from the the way tasks are chosen for Π_0 in line 2. Otherwise, since τ_j belongs to Π, and hence, by the assignment in line 6 to Π_1,

$$\tau_j \in \Gamma_0 \tag{A.22}$$

by the assignment in line 4. By lines 1 and 5, the execution cost of a task in Γ_0 is at least that of a task not in $(\Gamma \cup \Pi)$. Hence, by (A.22), and because $\tau_k \notin (\Gamma \cup \Pi)$,

$$e_j \geq e_k \tag{A.23}$$

holds. We consider the following two cases.

Case 1: $\tau_k \in \Gamma_1$**.** Since $\tau_k \in \tau'$, $\tau_k \notin \Gamma$. Therefore, by (A.14), $\tau_k \notin \Gamma_0 \wedge \tau_k \in \Pi_0$. Hence, since $\tau_k \in \Pi_0$ and $\tau_j \in \Gamma_0$ by (A.22), (and $\tau_k \in \Gamma_1$ and $\tau_j \in \Pi_1$), by lines 4 and 3, $e_k - b_k \geq e_j - b_j$ holds. Thus, $b_k \leq b_j + e_k - e_j$ holds, which by (A.23) implies $b_k \leq b_j$.

Case 2: $\tau_k \notin \Gamma_1$**.** Because the execution cost of a task in Γ_0 is at least that of a task not in that subset, and τ_k is not in Π (and hence not in Π_1), by (A.9),

$$\tau_k \notin \Gamma_0 \ \wedge \ \tau_k \notin \Pi_0. \tag{A.24}$$

Further, because τ_j is in Π (and hence in Π_1), and by (A.22) in Γ_0, (A.9) and (A.12) imply that there exists a τ_ℓ such that τ_ℓ is in Π_0 and in Γ_1. Therefore, the assignment in line 3 implies that $e_\ell - b_\ell \geq e_j - b_j$. Thus, $b_\ell \leq b_j + e_\ell - e_j$. Also, since τ_ℓ is in Π_0 while τ_j is in Γ_0, the assignments in lines 1 and 2 imply that $e_j \geq e_\ell$. Hence, it follows that $b_\ell \leq b_j$. Since τ_ℓ is in Π_0, and by (A.24), τ_k is in neither Π_0 nor Γ_0, $b_k \leq b_\ell$ holds, which by $b_\ell \leq b_j$ implies that $b_k \leq b_j$. ■

To show that the bound in Theorem 4.1 holds if the two subsets are as given by Algorithm Choose-Γ-Π, Lemmas 4.8 and 4.7 have to be shown to hold. Though these lemmas seem intuitive, formal proofs are quite lengthy, and have been omitted since the line of reasoning is similar to that used in Section A.2, where (4.1) is assumed to hold.

Appendix B

Derivation of a Schedulability Test for Global Preemptive EDF

In this appendix, we derive a sufficient schedulability test for sporadic task systems under g-EDF using a lag-based reasoning. The availability of three different schedulability tests and analyses for g-EDF (those reported in [61], [20], and [33]) prompted and motivated us to extend and apply the techniques used in Chapter 4 with the hope of arriving at yet another and perhaps improved test. However, our efforts did not lead to a new test, but to the discovery of an already known test, namely, $U_{sum}(\tau) \leq M - (M-1)u_{\max}(\tau)$, through an independent and a somewhat different approach. Yet, we consider the analysis that we have developed to be interesting in its own right. Furthermore, one important difference with respect to the previous analyses is that we show that any task system that passes the aforementioned test is schedulable on M processors even if its jobs may be early released. As shown in Chapter 8, a schedulability test for systems that may be early released can be used to show that, if practical considerations do not allow periods to be non-integral, then a tardiness bound of one quantum can be guaranteed for every task system that passes the test.

We present our lag-based analysis for g-EDF by proving the following theorem. Our approach is the same as that used in Chapter 4. We determine a lower bound on the total system lag, LAG, that is necessary for a deadline miss, an upper bound on LAG that is possible for a task system, and use these to arrive at a schedulability test.

Theorem B.1 *On M processors,* g-EDF *correctly schedules every sporadic task system τ with relative deadlines equal to periods (implicit deadlines) and whose jobs may be early released if $U_{sum}(\tau) \leq M - (M-1) \cdot u_{\max}(\tau)$ holds.*

Proof: Our proof is by contradiction. So, assume that the theorem does not hold. This assumption implies that there exist **(i)** a concrete instantiation τ of a non-concrete task system τ^N, for which $U_{sum}(\tau) = U_{sum}(\tau^N) \leq M - (M-1) \cdot u_{\max}(\tau)$ holds, and **(ii)** a g-EDF schedule $\mathcal{S}$ for τ such that the following hold.

(P1) One or more jobs of τ with deadlines at time t_d miss their deadlines in $\mathcal{S}$, and no job of τ with a deadline before t_d misses its deadline.

(P2) No concrete instantiation of τ^N satisfying (P1) releases fewer jobs than τ.

We claim the following based on (P1) and (P2).

Claim B.1 *The deadline of every job of τ is at or before t_d.*

Proof: Under g-EDF, jobs with deadlines later than t_d cannot interfere with those with deadlines at most t_d. Therefore, even if every job with a deadline later than t_d is removed from τ, there exists an g-EDF schedule in which every job that misses its deadline at t_d in $\mathcal{S}$ will still miss its deadline. This contradicts (P2). ■

Claim B.1, in turn, implies the following, establishing a lower bound on LAG that is necessary for τ to miss a deadline at t_d.

Claim B.2 $\mathsf{LAG}(\tau, t_d, \mathcal{S}) > 0$.

Proof: Let τ_i be a task in τ that does not have a job that misses its deadline at t_d. Then, by Claim B.1, no job of τ has a deadline after t_d, and hence, all jobs of τ_i complete executing by t_d in both $\mathcal{S}$ and PS_τ. Therefore, the total allocation to τ_i in $[0, t_d)$ is the same in both $\mathcal{S}$ and PS_τ, implying that $\mathsf{lag}(\tau_i, t_d, \mathcal{S}) = 0$.

Next, let τ_k be a task with a job that misses its deadline at t_d in $\mathcal{S}$. Then, since in PS_τ, all jobs of τ_k complete executing by t_d, the total allocation in $[0, t_d)$ to τ_k is higher in PS_τ than in $\mathcal{S}$. Hence, $\mathsf{lag}(\tau_k, t_d, \mathcal{S}) > 0$ holds.

Thus, by (4.10), we can conclude that $\mathsf{LAG}(\tau, t_d, \mathcal{S})$ is greater than zero. ■

Let $0 < t' \leq t_d$ denote the earliest instant in $[0, t_d)$ at which

$$\mathsf{LAG}(\tau, t', \mathcal{S}) > 0 \tag{B.1}$$

holds. By Claim B.2 and because $\mathsf{LAG}(\tau, 0, \mathcal{S}) = 0$, t' is well defined. We next turn to determining an upper bound on LAG for τ at t'.

By (4.10), the LAG of τ at t' is equal to the sum of the lags of all its tasks. Therefore, (B.1) implies that there exists at least one task in τ whose lag at t' is greater than zero. Let τ_h denote such a task. That is,

$$\mathsf{lag}(\tau_h, t', \mathcal{S}) > 0. \tag{B.2}$$

Because τ_h has a positive lag at t', at least one job of τ_h released before t' is pending at t'. Also, t_d is the earliest time that any job misses its deadline in $\mathcal{S}$ (by (P1)), and $t' \leq t_d$ holds. Therefore, exactly one job of τ_h released before t' can be pending at t'. Let $\tau_{h,\ell}$ denote the pending job of τ_h at t'.

Let t'' denote the release time of $\tau_{h,\ell}$. Then, because no job with deadline before t_d misses its deadline and $t'' < t' = t_d$ holds, all jobs of τ_h released before t'' complete executing by t'' in $\mathcal{S}$. All of these jobs complete executing by t'' in PS_τ as well. If the eligibility time of $\tau_{h,\ell}$ is before its release time (*i.e.*, $\tau_{h,\ell}$ is early released), then $\tau_{h,\ell}$ may have executed before t'', in which case, the lag of τ_h at t'' will be negative; else, the lag of τ_h at t'' is zero. Thus, in any case, the lag of $\tau_{h,\ell}$ at t'' is at most zero. Let B (resp., $\bar{B}$) denote the cumulative time in $[t'', t')$ in which all M processors are busy (resp., not all processors are busy) in $\mathcal{S}$. That is,

$$t' - t'' = B + \bar{B}. \tag{B.3}$$

Since $\tau_{h,\ell}$ is released at t'' and is pending at t', task τ_h executes at every non-busy instant in $[t'', t')$, *i.e.*, τ_h executes for at least $\bar{B}$ time units in $[t'', t')$. Therefore, the lag of τ_h at t' can be computed as follows.

$$\begin{aligned}
\mathsf{lag}(\tau_h, t', \mathcal{S}) &\leq \mathsf{lag}(\tau_h, t'', \mathcal{S}) + \mathsf{A}(\tau_h, \mathsf{PS}_\tau, t'', t') - \mathsf{A}(\tau_h, \mathcal{S}, t'', t') && \text{(by (4.11))} \\
&\leq \mathsf{A}(\tau_h, \mathsf{PS}_\tau, t'', t') - \mathsf{A}(\tau_h, \mathcal{S}, t'', t') \\
&\quad \text{(because, as discussed above, the } \mathsf{lag} \text{ of } \tau_h \text{ is at most zero at } t''\text{)} \\
&\leq (t' - t'') \cdot u_h - \mathsf{A}(\tau_h, \mathcal{S}, t'', t') && \text{(by (4.7))} \\
&\leq (B + \bar{B}) \cdot u_h - \bar{B} \\
&\quad \text{(by (B.3) and because } \tau_h \text{ executes for at} \\
&\quad \text{least for } \bar{B} \text{ time units in } [t'', t')\text{)} \\
&= B \cdot u_h + \bar{B} \cdot (u_h - 1) && \text{(B.4)}
\end{aligned}$$

By (B.4) above and (B.2), we have

$$\begin{aligned}
& B \cdot u_h + \bar{B} \cdot (u_h - 1) > 0 \\
\Rightarrow\ & \bar{B} < \frac{B \cdot u_h}{1 - u_h}.
\end{aligned} \tag{B.5}$$

Next, an upper bound on the LAG of τ at t' can be computed as follows.

$$
\begin{aligned}
\mathsf{LAG}(\tau, t', \mathcal{S}) &= \mathsf{LAG}(\tau, t'', \mathcal{S}) + \mathsf{A}(\tau, \mathsf{PS}_\tau, t'', t') - \mathsf{A}(\tau, \mathcal{S}, t'', t') && \text{(by (4.12))} \\
&\leq \mathsf{A}(\tau, \mathsf{PS}_\tau, t'', t') - \mathsf{A}(\tau, \mathcal{S}, t'', t') \\
&\quad \text{(because } t'' < t'\text{, hence } \mathsf{LAG} \text{ at } t'' \text{ is at most 0)} \\
&\leq (t' - t'') \cdot U_{sum} - \mathsf{A}(\tau, \mathcal{S}, t'', t') && \text{(by (4.8))} \\
&\leq (t' - t'') \cdot U_{sum} - (B \cdot M + \bar{B}) \\
&\quad \text{(because, in } [t'', t') \text{ in } \mathcal{S}\text{, } M \text{ tasks execute at every busy} \\
&\quad \text{instant, and at least task } \tau_h \text{ executes at every non-busy in-} \\
&\quad \text{stant)} \\
&= (B + \bar{B}) \cdot U_{sum} - B \cdot M - \bar{B} && \text{(by (B.3))} \\
&= B \cdot (U_{sum} - M) + \bar{B} \cdot (U_{sum} - 1)
\end{aligned}
$$

If $U_{sum} \leq 1$ holds, then the final expression above is at most zero, which contradicts (B.1). Hence, assume $U_{sum} > 1$ holds, in which case, the final expression above is an increasing function of $\bar{B}$. Therefore, by (B.5), we have

$$\mathsf{LAG}(\tau, t', \mathcal{S}) < B \cdot (U_{sum} - M) + \left(\frac{B \cdot u_h}{1 - u_h}\right) \cdot (U_{sum} - 1),$$

which by the fact that LAG at t' exceeds zero (as given by (B.1)), results in the following.

$$
\begin{aligned}
& B \cdot (U_{sum} - M) + \left(\frac{B \cdot u_h}{1 - u_h}\right) \cdot (U_{sum} - 1) > 0 \\
\Rightarrow\ & (U_{sum} - M) + \left(\frac{u_h}{1 - u_h}\right) \cdot (U_{sum} - 1) > 0 \\
\Rightarrow\ & (U_{sum} - M) \cdot (1 - u_h) + u_h \cdot (U_{sum} - 1) > 0 \\
\Rightarrow\ & U_{sum} > u_h + M(1 - u_h) \\
& \qquad = M - (M - 1) \cdot u_h \\
& \qquad \geq M - (M - 1) \cdot u_{\max}
\end{aligned}
$$

Thus, our assumption that $U_{sum} \leq M - (M - 1) \cdot u_{\max}$ holds is contradicted, from which the theorem follows. ∎

The above theorem is for implicit deadline systems. Let $\lambda_i = e_i / D_i$ denote the density of τ_i in a constrained deadline system (*i.e.*, a task system with relative deadlines at most periods) τ and let $\lambda_{\max}(\tau)$ denote the maximum density of any task in τ. Then, a schedulability test for systems with constrained deadlines is given by the following corollary. (An alternative proof is provided by Bertogna *et al.* in [33].)

Corollary B.1 *A sporadic task system with constrained deadlines is correctly scheduled by* g-EDF *on M processors if* $\sum_{i=1}^{n} \lambda_i \leq M - (M-1) \cdot \lambda_{\max}$ *holds.*

Proof: Let $\tau = \{\tau_i, 1 \leq i \leq n\}$ be a constrained deadline system and let τ' be an implicit deadline system constructed from τ as follows: $\tau' = \{\tau_i'(e_i', p_i') \mid 1 \leq i \leq n, e_i' = e_i, p_i' = D_i\}$. That is, there is a bijective mapping from τ to τ' such that for each task τ_i in τ, there is a corresponding task τ_i' in τ' whose execution cost is the same as that of τ_i and whose period is given by τ_i's relative deadline. Then, $u_i' = \lambda_i$ holds for all i and $U_{sum}(\tau') = \sum_{i=1}^{n} \lambda_i$ holds. Hence, by Theorem B.1, τ' is g-EDF-schedulable on M processors if $\sum_{i=1}^{n} \lambda_i \leq M - (M-1) \cdot \lambda_{\max}(\tau)$ holds. Now, every concrete instantiation of τ can be thought of as a concrete instantiation of τ' in which two consecutive job releases of τ_i' are separated by at least $p_i \geq D_i = p_i'$ time units. Since every concrete instance of τ' meets all its deadlines if the condition specified above is satisfied, every concrete instance of τ satisfying the condition can be correctly scheduled as well. ∎

Appendix C

Remaining Proofs from Chapter 6

Lemma 6.10 *For all $i \geq 1$, $k \geq 1$, the following holds.*

$$\mathsf{r}(T_{i+k}) \geq \begin{cases} \mathsf{d}(T_i) + k - 1, & \mathsf{b}(T_i) = 0 \\ \mathsf{d}(T_i) + k - 2, & \mathsf{b}(T_i) = 1 \end{cases}$$

Proof: We consider two cases based on $b(T_i)$.

Case 1: $\mathbf{b}(\boldsymbol{T_i}) = \mathbf{0}$. In this case, by (3.6), $\frac{i}{wt(T)}$ is an integer. Hence,

$$\begin{array}{rcll}
\mathsf{r}(T_{i+k}) & = & \Theta(T_{i+k}) + \left\lfloor \dfrac{i+k-1}{wt(T)} \right\rfloor & \text{(by (3.9))} \\
& = & \Theta(T_{i+k}) + \dfrac{i}{wt(T)} + \left\lfloor \dfrac{k-1}{wt(T)} \right\rfloor & \text{(because } \frac{i}{wt(T)} \text{ is an integer)} \\
& = & \Theta(T_{i+k}) + \left\lceil \dfrac{i}{wt(T)} \right\rceil + \left\lfloor \dfrac{k-1}{wt(T)} \right\rfloor & \\
& > & \Theta(T_{i+k}) + \left\lceil \dfrac{i}{wt(T)} \right\rceil + \dfrac{k-1}{wt(T)} - 1 & \\
& \geq & \Theta(T_i) + \left\lceil \dfrac{i}{wt(T)} \right\rceil + \dfrac{k-1}{wt(T)} - 1 & \text{(because } \Theta(T_{i+k}) \geq \Theta(T_i)\text{)} \\
& = & \mathsf{d}(T_i) + \dfrac{k-1}{wt(T)} - 1 & \text{(by (3.10))} \\
& \geq & \mathsf{d}(T_i) + k - 2 & \text{(because } wt(T) \leq 1\text{).}
\end{array}$$

Because, $\mathsf{r}(T_{i+k})$ is an integer, the above implies that $\mathsf{r}(T_{i+k}) \geq \mathsf{d}(T_i) + k - 1$.

Case 2: $\mathbf{b}(\boldsymbol{T_i}) = \mathbf{1}$. In this case,

$$\begin{array}{rcll}
\mathsf{r}(T_{i+k}) & = & \Theta(T_{i+k}) + \left\lfloor \dfrac{i+k-1}{wt(T)} \right\rfloor & \text{(by (3.9))} \\
& > & \Theta(T_{i+k}) + \dfrac{i+k-1}{wt(T)} - 1 & \\
& = & \Theta(T_{i+k}) + \dfrac{i}{wt(T)} + \dfrac{k-1}{wt(T)} - 1 &
\end{array}$$

$$
\begin{aligned}
&> \Theta(T_{i+k}) + \left\lceil \frac{i}{wt(T)} \right\rceil + \frac{k-1}{wt(T)} - 2 \\
&\geq \Theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil + \frac{k-1}{wt(T)} - 2 && \text{(because } \Theta(T_{i+k}) \geq \Theta(T_i)\text{)} \\
&= \mathsf{d}(T_i) + \frac{k-1}{wt(T)} - 2 && \text{(by (3.10))} \\
&\geq \mathsf{d}(T_i) + k - 3 && \text{(because } wt(T) \leq 1\text{).}
\end{aligned}
$$

For this case again, the lemma holds because $\mathsf{r}(T_{i+k})$ is an integer. ■

Lemma 6.2 $0 \leq \delta < \frac{1}{2}$.

Proof: By Definition 6.5, $\delta = \frac{\rho_{\max}}{1+\rho_{\max}}$. The first derivative of δ with respect to $\rho_{\max}$, $\delta'(\rho_{\max})$, is $\frac{1}{(1+\rho_{\max})^2}$, which is positive, and hence, δ is increasing with $\rho_{\max}$ (though the rate of increase decreases with $\rho_{\max}$). By (6.2) and (6.1), $0 \leq \rho_{\max} < W_{\max}$ holds. Hence, $\rho_{\max}$ lies in the range $[0, 1)$. $\delta(0) = 0$ and $\delta(1) = \frac{1}{2}$, from which, the lemma follows. ■

Lemma 6.7 *If* $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t-\ell)$, *where* $0 \leq \ell \leq \lambda-2$ *and* $t \geq \ell$, *then* $B(t-\ell, t+1) \neq \emptyset$.

Proof: We will refer to sets $A(t-\ell, t+1)$, $B(t-\ell, t+1)$, and $I(t-\ell, t+1)$ as A, B, and I, respectively. By (3.28), $\mathsf{LAG}(\tau, t+1) = \mathsf{LAG}(\tau, t-\ell) + \sum_{T \in \tau}(\mathsf{A}(\mathsf{PS}, T, t-\ell, t+1) - \mathsf{A}(\mathcal{S}, T, t-\ell, t+1))$. Because, tasks in I are neither active nor scheduled in the interval $[t-\ell, t+1)$, by (3.33), $\mathsf{LAG}(\tau, t+1) = \mathsf{LAG}(\tau, t-\ell) + \sum_{T \in A \cup B}(\mathsf{A}(\mathsf{PS}, T, t-\ell, t+1) - \mathsf{A}(\mathcal{S}, T, t-\ell, t+1))$ holds. Because $\ell \leq \lambda - 2$ holds by the statement of the lemma, $t+1-(t-\ell) \leq \lambda - 1$ holds. Therefore, by (3.16), for each T in $A \cup B$, $\mathsf{A}(\mathsf{PS}, T, t-\ell, t+1) \leq (\lambda-1) \cdot wt(T) \leq 1$ holds. The last inequality holds because, by (6.4), either $\lambda = 2$ or $W_{\max} \leq \frac{1}{\lambda-1}$. Hence, $\mathsf{LAG}(\tau, t+1) \leq \mathsf{LAG}(\tau, t-\ell) + |A \cup B| - \sum_{T \in A \cup B} \mathsf{A}(\mathcal{S}, T, t-\ell, t+1)$. By the statement of the lemma, $\mathsf{LAG}(\tau, t+1) > \mathsf{LAG}(\tau, t-\ell)$. Hence, it follows that $|A \cup B| - \sum_{T \in A \cup B} \mathsf{A}(\mathcal{S}, T, t-\ell, t+1) > 0$, *i.e.*,

$$\textstyle\sum_{T \in A \cup B} \mathsf{A}(\mathcal{S}, T, t-\ell, t+1) < |A \cup B|. \tag{C.1}$$

Now, if B is empty, then, $|A \cup B| = |A|$ holds. Because, by definition, every task in A is scheduled at least once in the interval $[t-\ell, t+1)$, $\sum_{T \in A \cup B} \mathsf{A}(\mathcal{S}, T, t-\ell, t+1) \geq |A| = |A \cup B|$, which is a contradiction to (C.1). Therefore, $B(t-\ell, t+1) \neq \emptyset$. ■

Lemma C.1 *Let T be a unit-weight task. Then,* $\mathsf{d}(T_i) = \mathsf{r}(T_i) + 1$ *and* $|\omega(T_i)| = 1$ *hold for every subtask T_i of T.*

Proof: Follows directly from $wt(T) = 1$, (3.9) and (3.10), and the definition of $\omega(T_i)$. ■

In some of the proofs that follow, we identify tasks that receive an allocation of zero at time t in the ideal schedule using the following definition.

Definition C.1: A GIS task U is a *Z-task* at time t iff there exists no subtask U_j such that $\mathsf{r}(U_j) \leq t < \mathsf{d}(U_j)$. The set of all tasks that are Z-tasks at t is denoted $Z(t)$.

By (3.15), and because $\mathsf{A}(\mathsf{PS}, U, t) = \sum_i \mathsf{A}(\mathsf{PS}, U_i, t)$ holds, U is a Z-task at t iff $\mathsf{A}(\mathsf{PS}, U, t) = 0$. Also, note that while $T \in I(t) \Rightarrow T \in Z(t)$, the converse does not hold. This is because, a task may have one or more subtasks whose IS-windows include slot t, and hence, are active at t, even if their release times are after t, *i.e.*, even if their PF-windows do not include slot t.

The next two lemmas concern tasks with unit weight that are Z-tasks at some time t. The first lemma says that if a subtask of a unit-weight task is scheduled before its release time, then that task is a Z-task during some earlier time.

Lemma C.2 *Let T_i be a subtask of a unit-weight task T scheduled at some time $t < t_d$. If $\mathsf{r}(T_i) > t$, then there exists at least one slot t' in $[0, t+1)$ such that T is a Z-task at t'.*

Proof: Follows easily from the fact that unit-weight tasks have PF-windows of length one. ■

Lemma C.3 *Let T be a unit-weight task and suppose T is not scheduled at some time $t < t_d$, where there is at least one hole in t. Then, there exists a time $t' \leq t$, such that there are no holes in $[t', t)$ and there exists at least one slot $\hat{t}$ in $[t', t+1)$, such that T is a Z-task in $\hat{t}$.*

Proof: Because T is not scheduled at t, by (3.33), either $T \in B(t)$ or $T \in I(t)$. If $T \in I(t)$, then T is a Z-task at t. Therefore, the lemma is trivially satisfied with $t' = t$. So assume $T \in B(t)$ for the rest of the proof. Let T_j be the critical subtask at t of T. Then, by Definition 3.2, we have

$$\mathsf{d}(T_j) \geq t + 1. \tag{C.2}$$

Also, because there is a hole in t, by Lemma 6.4, T_j is scheduled before t, say t'', *i.e.*,

$$t'' < t \ \wedge \ \mathcal{S}(T_j, t'') = 1. \tag{C.3}$$

Because there is a hole in t, by Lemma 6.5, $\mathsf{d}(T_j) \leq t+1$ holds, which, by (C.2), implies that $\mathsf{d}(T_j) = t+1$ holds. Because $wt(T) = 1$, by Lemma C.1, we have $\mathsf{r}(T_j) = t$. Thus,

$$\mathsf{r}(T_j) = t \ \wedge \ \mathsf{d}(T_j) = t+1. \tag{C.4}$$

Because (C.3) and (C.4) hold, by Lemma C.2, T is a Z-task at some time before t. Let z be the latest such time. Then, we have

$$\begin{aligned} &0 \leq z < t \ \wedge \ (\nexists T_k : \mathsf{r}(T_k) \leq z \ \wedge \ \mathsf{d}(T_k) > z) \\ &\quad \wedge \ (\forall z' : z < z' \leq t :: (\exists T_k : \mathsf{r}(T_k) \leq z' \ \wedge \ \mathsf{d}(T_k) > z')). \end{aligned} \tag{C.5}$$

Because $wt(T) = 1$ holds, by Lemma C.1, (C.5) implies the following.

$$\begin{aligned} &0 \leq z < t \ \wedge \ (\nexists T_k : \mathsf{r}(T_k) = z \wedge \ \mathsf{d}(T_k) = z+1) \\ &\quad \wedge \ (\forall z' : z < z' \leq t :: (\exists T_k : \mathsf{r}(T_k) = z' \ \wedge \ \mathsf{d}(T_k) = z'+1)) \end{aligned} \tag{C.6}$$

We next claim the following. (Informally, Claim C.1 says that every subtask of T with release time in $[z+1, t]$ is scheduled before its release time, and there is no hole in any slot between where the subtask is scheduled and its deadline.)

Claim C.1 *If* $\mathsf{r}(T_k) = z' \ \wedge \ \mathsf{d}(T_k) = z'+1$ *holds, where* $z < z' \leq t$*, then there exists a* u'*, where* $u' \leq t'' - (t - z') < z'$*, such that* T_k *is scheduled at* u'*, and there is no hole in any slot in* $[u', z')$*.*

The proof is by induction on decreasing values for z'. So, we begin with $z' = t$.

Base Case: $z' = t$**.** By (C.4), we have $\mathsf{r}(T_j) = t$ and $\mathsf{d}(T_j) = t+1$, and by (C.3), we have $\mathcal{S}(T_j, t'') = 1$. Further, $t'' - (t - z') = t'' - (t-t) = t''$. By (C.3), $t'' < t = z'$. Therefore, to show that the claim holds for this case with $u' = t''$, we only have to show that there is no hole in any slot in $[t'', t)$. Assume to the contrary that there is a hole in slot t_h in $[t'', t)$. Then, by Lemma 6.5, $\mathsf{d}(T_j) \leq t_h + 1 < t+1$, which is in contradiction to (C.4). Hence, the assumption does not hold.

Induction Hypothesis: Assume that for all z', where $z'' \leq z' \leq t$ and $t \geq z'' > z+1$, $(\mathsf{r}(T_k) = z' \ \wedge \ \mathsf{d}(T_k) = z'+1) \Rightarrow \ (\exists u' : u' \leq t'' - (t - z') < z' :: (\mathcal{S}(T_k, u') = 1 \ \wedge \ \text{there are no holes in } [u', z'))$ holds. (Informally, the claim holds for all z' in $[z'', t]$.)

Induction Step: We now show that the following holds, *i.e.*, the claim holds, for $z'' - 1$.

$$(\mathsf{r}(T_k) = z'' - 1 \ \wedge \ \mathsf{d}(T_k) = z'') \Rightarrow (\exists u' : u' \leq t'' - (t - (z'' - 1)) < z'' :: \\ (\mathcal{S}(T_k, u') = 1 \ \wedge \ \text{there is no hole in any slot in } [u', z'' - 1)) \qquad \text{(C.7)}$$

By (C.6), there exists a subtask T_ℓ with $\mathsf{r}(T_\ell) = z''$ and $\mathsf{d}(T_\ell) = z'' + 1$. By the induction hypothesis, T_ℓ is scheduled at or before $t'' - (t - z'')$. Let T_k be a subtask such that $\mathsf{r}(T_k) = z'' - 1$ and $\mathsf{d}(T_k) = z''$. Then, because $\mathsf{r}(T_k) < \mathsf{r}(T_\ell)$ holds, $k < \ell$ holds. Also, $\mathsf{d}(T_k) < \mathsf{d}(T_\ell)$. Hence, by (3.12), T_k is scheduled before T_ℓ, *i.e.*, T_k is scheduled at or before $t'' - (t - z'') - 1 = t'' - (t - (z'' - 1))$. Because $t'' - (t - (z'' - 1)) = z'' - 1 - (t - t'')$, by (C.3), we have $t'' - (t - (z'' - 1)) < z'' - 1$. Thus, we have shown that if the antecedent of the implication in (C.7) holds, then the first subexpression of the consequent is satisfied. To see that there are no holes in $[t'' - (t - (z'' - 1)), z'' - 1)$, assume to the contrary that there is a hole in slot t_h in this interval. Then, by Lemma 6.5, $\mathsf{d}(T_k) \leq t_h + 1 \leq z'' - 1$, which is in contradiction to $\mathsf{d}(T_k) = z''$. Thus, the claim holds for $z'' - 1$. ■

(C.6) and Claim C.1 imply that $(\forall z' : z < z' \leq t ::$ there are no holes in $z' - 1)$. Therefore, it immediately follows that there are no holes in $[z, t)$. Also, we defined z such that T is a Z-task at z. Therefore, the lemma follows. ■

Proof of parts (f), (g), (h), and (i) of Lemma 6.8.

Lemma 6.8 *The following properties hold for τ and $\mathcal{S}$.*

(a) *For all T_i in τ, $\mathsf{d}(T_i) \leq t_d$.*

(b) *Exactly one subtask of τ misses its deadline at t_d.*

(c) $\mathsf{LAG}(\tau, t_d) = 1$.

(d) *Let T_i be a subtask in τ that is scheduled at $t < t_d$ in $\mathcal{S}$. Then, $\mathsf{e}(T_i) = \min(\mathsf{r}(T_i), t)$.*

(e) $(\forall T_i \in \tau :: \mathsf{d}(T_i) < t_d \ \Rightarrow \ (\exists t :: \mathsf{e}(T_i) \leq t < \mathsf{d}(T_i) \ \wedge \ \mathcal{S}(T_i, t) = 1))$. *That is, every subtask with deadline before t_d is scheduled before its deadline.*

(f) *Let U_k be the subtask in τ that misses its deadline at t_d. Then, U is not scheduled in any slot in $[t_d - \lambda + 1, t_d)$.*

(g) *There is no hole in any of the last $\lambda - 1$ slots,* i.e., *in slots in $[t_d - \lambda + 1, t_d)$.*

(h) *There exists a time* $v \le t_d - \lambda$ *such that the following both hold.*

(**i**) *There is no hole in any slot in* $[v, t_d - \lambda)$.

(**ii**) $\mathsf{LAG}(\tau, v) \ge (t_d - v)(1 - \alpha)M + 1$.

(i) *There exists a time* $u \in [0, v)$ *where* v *is as defined in part (h), such that* $\mathsf{LAG}(\tau, u) < 1$ *and* $\mathsf{LAG}(\tau, t) \ge 1$ *for all* t *in* $[u + 1, v]$.

As mentioned in the main text, parts (a), (b), (c), and (d) are proved in [105]. Part (e) follows directly from Definition 6.2 and (T1). The other parts are proved below.

Proof of (f): If U_k is the first subtask of U, then it is trivial that U is not scheduled at any time before t_d. So assume U_k is not the first subtask of U and let U_j be U_k's predecessor. We first show that the deadline of U_j is at or before $t_d - \lambda + 1$. Because U_k misses its deadline at t_d, $\mathsf{d}(U_k) = t_d$ holds. If $W_{\max} < 1$, then by Lemma 6.9, $|\omega(U_k)| \ge \lambda$. On the other hand, if $W_{\max} = 1$, then by (6.4), $\lambda = 2$. Hence, if $wt(U) < 1$, then by Lemma 3.1, $|\omega(U_k)| \ge 2 = \lambda$. Thus, if either $W_{\max} < 1$ or $W_{\max} = 1 \ \wedge \ wt(U) < 1$, then $\mathsf{r}(U_k) \le t_d - \lambda$ holds. Since U_k is U_j's predecessor, $k \le j - 1$ holds, and by Lemma 6.10, $\mathsf{d}(U_j) \le t_d - \lambda + 1$ follows. Finally, if $W_{\max} = 1$ and $wt(U) = 1$, then because the deadlines of any two consecutive subtasks of a unit-weight task are separated by at least one time slot, $\mathsf{d}(U_j) \le t_d - 1 = t_d - \lambda + 1$ follows.

Thus, by part (e), U_j and every prior subtask of U is scheduled before $t_d - \lambda + 1$. Obviously, no U_ℓ, where $\ell > k$, is scheduled before U_k is scheduled. Therefore, no subtask of U is scheduled in any slot in $[t_d - \lambda + 1, t_d)$. ■

Proof of (g): Let U_k denote the subtask that misses its deadline at t_d. By Lemma 6.9, $|\omega(U_k)| \ge \lambda - 1$ holds, Therefore, $\mathsf{r}(U_k) \le t_d - \lambda + 1$. Further, by part (f), no subtask of U is scheduled in any slot in $[t_d - \lambda + 1, t_d)$. Hence, U_k is ready and eligible to be scheduled in $[t_d - \lambda + 1, t_d)$. Thus, if there is a hole in this interval, then EPDF would schedule U_k there, which contradicts the fact that U_k misses its deadline. ■

Proof of (h): Let τ_1 and N_1 be defined as follows.

$$\tau_1 \stackrel{\text{def}}{=} \{T \mid T \in \tau \ \wedge \ wt(T) = 1\} \tag{C.8}$$

$$|\tau_1| \stackrel{\text{def}}{=} N_1 \tag{C.9}$$

Then, by (T0), we have

$$N_1 \le M, \tag{C.10}$$

and by Definition 6.4,

$$\sum_{T \in \tau \setminus \tau_1} wt(T) = \alpha M - N_1. \tag{C.11}$$

By part (g), there is no hole in any slot in $[t_d - \lambda + 1, t_d)$. Therefore, by (C.10), at least $M - N_1$ tasks with weight less than one (*i.e.*, tasks in $\tau \setminus \tau_1$) are scheduled in each slot in $[t_d - \lambda + 1, t_d)$, *i.e.*,

$$\left(\forall t : t_d - \lambda + 1 \leq t < t_d :: \sum_{T \in \tau \setminus \tau_1} \mathcal{S}(T, t) \geq M - N_1 \right). \tag{C.12}$$

Let $\mathcal{T}$ denote the set of all subtasks of tasks not in τ_1 that are scheduled in some slot in $[t_d - \lambda + 1, t_d)$, *i.e.*,

$$\mathcal{T} = \{T_i \mid T \in \tau \setminus \tau_1 \ \wedge \ T_i \text{ is scheduled in the interval } [t_d - \lambda + 1, t_d)\}. \tag{C.13}$$

Let T_i be some subtask in $\mathcal{T}$. If $W_{\max} < 1$, then by Lemma 6.9, $|\omega(T_i)| \geq \lambda$ holds. Otherwise, since, $T \notin \tau_1$, $wt(T) < 1$ holds, and hence, by Lemma 3.1, $|\omega(T_i)| \geq 2 = \lambda$ holds. (The last equality follows from (6.4), since $W_{\max} = 1$.) Thus, in both cases, the length of the PF-window of every subtask in $\mathcal{T}$ is at least λ. By part (a), $\mathsf{d}(T_i) \leq t_d$. Hence, because $|\omega(T_i)| \geq \lambda$ holds, it follows that $\mathsf{r}(T_i) \leq t_d - \lambda$. Therefore, by Lemma 6.10, the deadline of the predecessor, if any, say, T_h, of T_i is at or before $t_d - \lambda + 1$. Hence, by part (e), T_h and all prior subtasks of T are scheduled before $t_d - \lambda + 1$. Thus, at most one subtask of every task not in τ_1 is scheduled in $[t_d - \lambda + 1, t_d)$. Furthermore, as shown above, the release time of each such subtask is at or before $t_d - \lambda$.

By (C.12), $|\mathcal{T}| \geq (\lambda - 1) \cdot (M - N_1)$, and by the discussion above, every subtask in $\mathcal{T}$ is of a distinct task, and the release time of each such subtask is at or before $t_d - \lambda$. By (C.9), in every time slot, at least $M - N_1$ processors are available for scheduling tasks not in τ_1. Therefore, the fact that no subtask in $\mathcal{T}$ is scheduled at $t_d - \lambda$ implies that either there is no hole in $t_d - \lambda$, that is, M other subtasks of tasks in $\tau \setminus \tau_1$ are scheduled there, or there is a hole in $t_d - \lambda$ and the predecessor of every subtask in $\mathcal{T}$ (that is at least $(\lambda - 1) \cdot (M - N_1)$ distinct tasks in $\tau \setminus \tau_1$) is scheduled there. Therefore, we have the following.

(P1) At least $\min(M, (\lambda - 1) \cdot (M - N_1))$ tasks in $\tau \setminus \tau_1$ are scheduled at $t_d - \lambda$.

Let $h \geq 0$ denote the number of holes in slot $t_d - \lambda$. If $h > 0$ holds, then by (P1), $(\lambda-1)\cdot(M - N_1) < M$ holds, and at least $(\lambda-1)\cdot(M-N_1)$ tasks in $\tau \setminus \tau_1$ are scheduled in $t_d - \lambda$. Hence, at most $M - (\lambda-1)\cdot(M-N_1) - h = (\lambda-1)N_1 - (\lambda-2)M - h$ tasks in τ_1 are scheduled at $t_d - \lambda$. Let

$$N_1^s \stackrel{\text{def}}{=} \sum_{T \in \tau_1} \mathcal{S}(T, t_d - \lambda). \quad \text{(C.14)}$$

Then, by the above discussion,

$$h > 0 \Rightarrow N_1^s \leq (\lambda-1)N_1 - (\lambda-2)M - h < (\lambda-1)N_1 - (\lambda-2)M. \quad \text{(C.15)}$$

Let τ_1^z denote the subset of all tasks in τ_1 that are not scheduled at $t_d - \lambda$. If τ_1^z is not empty and $h > 0$, then let Y be a task in τ_1^z. Then, by Lemma C.3, there exists a time $u \leq t_d - \lambda$ such that Y is a Z-task at u and there is no hole in any slot in $[u, t_d - \lambda)$. Let v be defined as follows.

$$v \stackrel{\text{def}}{=} \begin{cases} t_d - \lambda, & \text{if } \tau_1^z = \emptyset \ \vee \ h = 0 \\ \min_{Y \in \tau_1^z}\{u \leq t_d - \lambda \mid \text{there is no hole in } [u, t_d - \lambda) \text{ and } \\ \quad Y \text{ is a } Z\text{-task in at least one slot in } [u, t_d - \lambda + 1)\}, & \text{if } \tau_1^z \neq \emptyset \ \wedge \ h > 0 \end{cases}$$

v satisfies the following.

Every task in τ_1^z is a Z-task in at least one slot in the interval $[v, t_d - \lambda + 1)$. (C.16)

There is no hole in $[v, t_d - \lambda)$. (C.17)

To complete the proof, we are left with determining a lower bound on $\mathsf{LAG}(\tau, v)$. By (3.28), we have

$$\begin{aligned}
&\mathsf{LAG}(\tau, v) \\
&= \mathsf{LAG}(\tau, t_d) - \textstyle\sum_{T \in \tau} \sum_{u=v}^{t_d - 1}(\mathsf{A}(\mathsf{PS}, T, u) - \mathcal{S}(T, u)) && \text{(by (3.28))} \\
&= 1 - \textstyle\sum_{T \in \tau} \sum_{u=v}^{t_d - 1}(\mathsf{A}(\mathsf{PS}, T, u) - \mathcal{S}(T, u)) && \text{(by part (c))} \\
&= 1 - \textstyle\sum_{T \in \tau} \sum_{u=v}^{t_d - 1}\mathsf{A}(\mathsf{PS}, T, u) + \sum_{T \in \tau} \sum_{u=v}^{t_d - 1}\mathcal{S}(T, u) \\
&= 1 - (\textstyle\sum_{T \in \tau} \sum_{u=v}^{t_d - 1}\mathsf{A}(\mathsf{PS}, T, u)) + (t_d - v)M - h \\
&\qquad \text{(there are } h \text{ holes in } t_d - \lambda\text{; by part (g), there are no holes in } [t_d - \lambda + 1, t_d), \\
&\qquad \text{and by (C.17), there is no hole in } [v, t_d - \lambda)) \\
&= 1 - \textstyle\sum_{T \in \tau_1} \sum_{u=v}^{t_d - \lambda}\mathsf{A}(\mathsf{PS}, T, u) - \sum_{T \in \tau \setminus \tau_1} \sum_{u=v}^{t_d - \lambda}\mathsf{A}(\mathsf{PS}, T, u) \\
&\quad - \textstyle\sum_{T \in \tau} \sum_{u=t_d - \lambda + 1}^{t_d - 1}\mathsf{A}(\mathsf{PS}, T, u) + (t_d - v)M - h
\end{aligned}$$

$$
\begin{aligned}
&\geq \; 1 - \textstyle\sum_{T\in\tau_1}\sum_{u=v}^{t_d-\lambda} \mathsf{A}(\mathsf{PS},T,u) - \sum_{T\in\tau\setminus\tau_1}\sum_{u=v}^{t_d-\lambda} wt(T) \\
&\quad - \textstyle\sum_{T\in\tau}\sum_{u=t_d-\lambda+1}^{t_d-1} wt(T) + (t_d - v)M - h && \text{(by (3.16))} \\
&= \; 1 - \textstyle\sum_{T\in\tau_1}\sum_{u=v}^{t_d-\lambda} \mathsf{A}(\mathsf{PS},T,u) \\
&\quad -(\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h \\
&&& \text{(by (C.11) and Def. 6.4)} \\
&= \; 1 - \textstyle\sum_{T\in(\tau_1\setminus\tau_1^z)}\sum_{u=v}^{t_d-\lambda} \mathsf{A}(\mathsf{PS},T,u) - \sum_{T\in\tau_1^z}\sum_{u=v}^{t_d-\lambda} \mathsf{A}(\mathsf{PS},T,u) \\
&\quad -(\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h \\
&\geq \; 1 - \textstyle\sum_{T\in(\tau_1\setminus\tau_1^z)}\sum_{u=v}^{t_d-\lambda} wt(T) - \sum_{T\in\tau_1^z}\sum_{u=v}^{t_d-\lambda} \mathsf{A}(\mathsf{PS},T,u) \\
&\quad -(\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h && \text{(by (3.16))} \\
&= \; 1 - (N_1^s)(t_d - v - \lambda + 1) - \textstyle\sum_{T\in\tau_1^z}\sum_{u=v}^{t_d-\lambda} \mathsf{A}(\mathsf{PS},T,u) \\
&\quad -(\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h \\
&&& \text{(by the definitions of } \tau_1 \text{ and } \tau_1^z\text{, (C.9), and (C.14))} \\
&= \; 1 - \textstyle\sum_{T\in\tau_1^z}\sum_{\{u \,|\, v \leq u \leq t_d - \lambda \;\wedge\; T \text{ is not a } Z\text{-task at } u\}} \mathsf{A}(\mathsf{PS},T,u) \\
&\quad -(N_1^s)(t_d - v - \lambda + 1) - (\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h \\
&&& \text{(by Def. C.1 (} Z\text{-task) and (3.15))} \\
&\geq \; 1 - \textstyle\sum_{T\in\tau_1^z}\sum_{\{u \,|\, v \leq u \leq t_d - \lambda \;\wedge\; T \text{ is not a } Z\text{-task at } u\}} wt(T) \\
&\quad -(N_1^s)(t_d - v - \lambda + 1) - (\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h \\
&&& \text{(by (3.16))} \\
&\geq \; 1 - \textstyle\sum_{T\in\tau_1^z}(t_d - v - \lambda)wt(T) - (N_1^s)(t_d - v - \lambda + 1) \\
&\quad -(\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h && \text{(by (C.16))} \\
&= \; 1 - (N_1 - N_1^s)(t_d - v - \lambda) - (N_1^s)(t_d - v - \lambda + 1) \\
&\quad -(\alpha M - N_1)(t_d - v - \lambda + 1) - (\lambda - 1)\alpha M + (t_d - v)M - h \\
&&& \text{(by the definitions of } \tau_1 \text{ and } \tau_1^z\text{, (C.9), and (C.14))} \\
&= \; 1 + N_1 - N_1^s - \alpha M(t_d - v) + M(t_d - v) - h && \text{(simplifying)} \\
&\geq \; \begin{cases} 1 + N_1 - N_1^s - \alpha M(t_d - v) + M(t_d - v), & h = 0 \\ 1 - (\lambda - 2)N_1 + (\lambda - 2)M - \alpha M(t_d - v) + M(t_d - v), & h > 0 \end{cases} \\
&&& \text{(by (C.15))} \\
&\geq \; 1 + (M - \alpha M)(t_d - v) \\
&&& \text{(because } N_1 - N_1^s \geq 0\text{, and, by (C.9) and (C.10), } N_1 \leq M\text{).}
\end{aligned}
$$

■

Proof of (i): Follows from the facts that $\mathsf{LAG}(\tau, 0) = 0$ and there exists a $v \leq t_d - \lambda$ such that $LAG(\tau, v) \geq (t_d - v)(1 - \alpha)M + 1$ (from part (h)). Note that, by Lemma 6.1,

$(t_d - v)(1-\alpha)M + 1 \geq 1$. ■

Lemma 6.15 *A solution to the recurrence*

$$\begin{aligned} L_0 &< \delta + \delta \cdot \alpha M \\ L_k &\leq \delta \cdot L_{k-1} + \delta \cdot (\lambda\alpha M - (\lambda-1)M), \end{aligned} \tag{C.18}$$

for all $k \geq 1$ and $0 \leq \delta < 1$, is given by

$$L_n < \delta^{n+1}(1+\alpha M) + (1-\delta^n)\left(\frac{\delta}{1-\delta}\right)(\lambda\alpha M - (\lambda-1)M), \quad \textit{for all } n \geq 0. \tag{C.19}$$

Proof: The proof is by induction on n.

Base Case: Holds trivially for $n = 0$.

Induction Hypothesis: Assume that (C.19) holds for $L_0, \ldots, L_k$.

Induction Step: Using (C.18), we have

$$\begin{aligned} L_{k+1} &\leq \delta L_k + \delta(\lambda\alpha M - (\lambda-1)M) \\ &< \delta^{k+2}(1+\alpha M) + \delta(1-\delta^k)\left(\tfrac{\delta}{1-\delta}\right)(\lambda\alpha M - (\lambda-1)M) + \delta(\lambda\alpha M - (\lambda-1)M) \\ &\qquad\qquad \text{(by (C.19) (induction hypothesis))} \\ &= \delta^{k+2}(1+\alpha M) + \delta(\lambda\alpha M - (\lambda-1)M)((1-\delta^k)\left(\tfrac{\delta}{1-\delta}\right) + 1) \\ &= \delta^{k+2}(1+\alpha M) + (1-\delta^{k+1})\left(\tfrac{\delta}{1-\delta}\right)((\lambda\alpha M - (\lambda-1)M), \end{aligned}$$

which proves the lemma. ■

Lemma 6.16 $\frac{M(\lambda-\delta-(\lambda-1)\delta^{n+1})+\delta^{n+2}-\delta^{n+1}+1-\delta}{M(\lambda-(\lambda-1)\delta^{n+1}-\delta^{n+2})} \geq \frac{M(\lambda-\delta)+\frac{1}{\lambda}}{\lambda M}$ *holds for all $n \geq 0, 0 \leq \delta \leq 1/2$, and $M \geq 1$.*

Proof: First, note the following.

$$\begin{aligned} &\tfrac{M(\lambda-\delta-(\lambda-1)\delta^{n+1})+\delta^{n+2}-\delta^{n+1}+1-\delta}{M(\lambda-(\lambda-1)\delta^{n+1}-\delta^{n+2})} \geq \tfrac{M(\lambda-\delta)+\frac{1}{\lambda}}{\lambda M} \\ \Leftrightarrow\ & (\lambda^2+1)\delta^{n+2} - (\lambda^2-\lambda+1)\delta^{n+1} + \lambda M\delta^{n+2}(1-\delta) + \lambda^2(1-\delta) - \lambda \geq 0 \\ &\qquad\qquad \text{(simplifying)} \\ \Leftarrow\ & (\lambda^2+1)\delta^{n+2} - (\lambda^2-\lambda+1)\delta^{n+1} + \lambda\delta^{n+2}(1-\delta) + \lambda^2(1-\delta) - \lambda \geq 0 \\ &\qquad\qquad \text{(because } M \geq 1 \text{ and } 0 \leq \delta \leq 1/2) \\ \Leftrightarrow\ & -\lambda\delta^{n+3} + (\lambda^2+\lambda+1)\delta^{n+2} - (\lambda^2-\lambda+1)\delta^{n+1} - \lambda^2\delta + \lambda^2 - \lambda \geq 0 \end{aligned}$$

From this, it suffices to show that $h(\delta,\lambda) \stackrel{\text{def}}{=} -\lambda\delta^{n+3} + (\lambda^2+\lambda+1)\delta^{n+2} - (\lambda^2-\lambda+1)\delta^{n+1} - \lambda^2\delta + \lambda^2 - \lambda \geq 0$, for $0 \leq \delta \leq 1/2$, $n \geq 0$, and $\lambda \geq 2$. The first derivative of $h(\delta,\lambda)$ with respect to λ is given by $h'(\delta, \lambda) = \delta^{n+2}(1-\delta) + (1-\delta^{n+1})(2\lambda(1-\delta)-1)$. Because $\delta \leq 1/2$ and $\lambda \geq 2$, $h'(\delta, \lambda)$ is positive for all λ. Hence, $h(\delta, \lambda)$ is increasing with λ and it suffices to show that $h(\delta, 2) \geq 0$ holds. $h(\delta, 2) = -2\delta^{n+3} + 7\delta^{n+2} - 3\delta^{n+1} - 4\delta + 2 = -(2\delta-1)(\delta^{n+1}(\delta-3)+2)$. Because $-(2\delta - 1) \geq 0$, it suffices to show that $g(\delta) \stackrel{\text{def}}{=} \delta^{n+1}(\delta - 3) + 2$ is at least zero. The first derivative of $g(\delta)$ with respect to δ is given by $g'(\delta) = \delta^n((n+2)\delta - 3(n+1))$. The roots of $g'(\delta)$ are $\delta = 0$ and $\delta = \frac{3(n+1)}{n+2}$. $\frac{3(n+1)}{n+2} \geq \frac{3}{2}$ holds for all $n \geq 0$. Therefore, $g(\delta)$ is either increasing or decreasing for δ in $[0, \frac{1}{2}]$. $g(0) = 2$ and $g(\frac{1}{2})$ lies in $[\frac{3}{4}, 2]$ for all $n \geq 0$. Therefore, $g(\delta)$ is positive in $[0, \frac{1}{2}]$, and hence, $h(\delta) \geq 0$ in that interval. ■

Lemma 6.6 *Let $t < t_d$ be a slot with holes and let U_j be a subtask that is scheduled at t. Then $\mathsf{d}(U_j) = t + 1$ and $\mathsf{b}(U_j) = 1$.*

Proof: Let U_j and t be as defined in the statement of the lemma. We first show that $\mathsf{d}(U_j)$ cannot be less than $t+1$. Because U_j is scheduled at t and $t \leq t_d - 1$ holds, $\mathsf{d}(U_j) < t+1 \Rightarrow \mathsf{d}(U_j) < t_d$. Hence, by Lemma 6.8(e), U_j is scheduled before its deadline, and so, $\mathsf{d}(U_j) < t+1$ implies that U_j is scheduled before t which contradicts the fact that U_j is scheduled at t. Therefore,

$$\mathsf{d}(U_j) \geq t + 1. \tag{C.20}$$

We now show that $\mathsf{d}(U_j) = t+1 \;\wedge\; \mathsf{b}(U_j) = 1$ holds. The proof is by contradiction, hence, assume that $\mathsf{b}(U_j) = 0 \;\vee\; \mathsf{d}(U_j) \neq t+1$ holds. By (C.20), this implies that $(\mathsf{b}(U_j) = 0 \;\wedge\; \mathsf{d}(U_j) = t+1) \;\;\vee\; \mathsf{d}(U_j) > t+1$. Let U_k be the successor, if any, of U_j. By Lemma 6.10, if $\mathsf{d}(U_j) > t+1$ holds, then $\mathsf{r}(U_k) \geq \mathsf{d}(U_j) - 1 > t$ holds, and if $\mathsf{b}(U_j) = 0 \wedge \mathsf{d}(U_j) = t+1$ holds, then $\mathsf{r}(U_k) \geq \mathsf{d}(U_j) = t+1$ holds. Thus, in both cases, $\mathsf{r}(U_k) \geq t+1$ holds. Because U_j is scheduled at t in $\mathcal{S}$, U_k is scheduled at or after $t+1$. Hence, by Lemma 6.8(d), $\mathsf{e}(T_k) \geq t+1$ holds. Now, if U_j is removed, then because there is at least a hole in t, by Lemma 3.6, only U_j's successor, U_k, can shift to t. However, if U_k exists, then its eligibility time is at least $t+1$, and hence, U_k cannot shift to t. Thus, regardless of whether U_k exists, the removal of U_j will not cause any subtask scheduled after t to shift to t, and hence, will not prevent the deadline miss at t_d. This contradicts (T2). Hence, our assumption is incorrect. The lemma follows. ■

Bibliography

[1] AMD multi-core. http://multicore2.amd.com/whatismc. 3

[2] Intel multi-core platforms. http://www.intel.com/technology/computing/multi-core/. 3

[3] T. Abdelzaher, B. Andersson, J. Jonsson, V. Sharma, and M. Nguyen. The aperiodic multiprocessor utilization bound for liquid tasks. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, pages 173–184, September 2002. 334

[4] T. Abdelzaher and C. Lu. Schedulability analysis and utilization bounds for highly scalable real-time services. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 15–25, June 2001. 334

[5] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, December 1998. 51

[6] L. Abeni and G. Buttazzo. QoS guarantees using probabilistic deadlines. In *Proceedings of the 11th Euromicro Conference of Real-Time Systems*, pages 242–249, June 1999. 51

[7] L. Abeni and G. Buttazzo. Stochastic analysis of a reservation based system. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 946–952, April 2001. 51

[8] J. Anderson and S. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, pages 428–435. IEEE, March 2004. 3

[9] J. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 130–141, December 2003. 76

[10] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, July 2005. 126

[11] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190. April 2006. 333

[12] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997. 77, 78

[13] 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. 67

[14] 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. 33, 66, 68, 73

[15] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, February 2004. 38, 59, 63, 73, 74

[16] B. Andersson, S. Baruah, and J. Jonsson. Static priority scheduling on multiprocessors. In *Proceedings of the 22nd Real-Time Systems Symposium*, pages 193–202, December 2001. 22, 85

[17] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 33–40, July 2003. 22

[18] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 123–132, December 1998. 50, 51

[19] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130, February 2001. 47

[20] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129, December 2003. 84, 85, 87, 349

[21] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 200–209, December 1995. 75

[22] S. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, June 2004. 22, 84, 184

[23] S. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1-2):9–20, February 2006. 85

[24] S. Baruah and J. Carpenter. Multiprocessor fixed-priority scheduling with restricted inter-processor migrations. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 195–202, July 2003. 21, 22

[25] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996. 16, 22, 33, 57, 59, 66, 73, 76, 90, 135

[26] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995. 73

[27] S. Baruah, J. Goossens, , and G. Lipari. Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 154–163, September 2002. 54

[28] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 100–110, October 1991. 42

[29] S. Baruah and G. Lipari. Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 109–116, July 2004. 54

[30] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium*, April 2004. 54

[31] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 328–335, December 2004. 48

[32] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001. 45

[33] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, July 2005. 85, 87, 349, 352

[34] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, December 2005. 85

[35] A. Block and J. Anderson. Accuracy versus migration overhead in multiprocessor reweighting algorithms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364, July 2006. 35

[36] A. Block, J. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 429–435, August 2005. 35, 76

[37] A. Block, J. Anderson, and U. Devi. Task reweighting under global scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 128–138, July 2006. 35, 119

[38] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973. 112

[39] G. Buttazzo. Rate Monotonic vs. EDF: Judgement day. *Real-Time Systems*, 29(1):5–26, January 2005. 17

[40] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 295–304, November 2000. 48

[41] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{rt}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, December 2006. To Appear. 37, 39, 268

[42] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004. 15, 20, 21, 22, 23, 25, 126

[43] 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 USENIX Symposium on Operating System Design and Implemetation*, pages 45–58, June 2000. 76

[44] C. Chetto and M. Chetto. Some results of the Earliest Deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989. 48

[45] R. Davis and A. Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 100–109, December 1995. 48

[46] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of IFIP Cong.*, pages 807–813, 1974. 14

[47] M. Dertouzos and A. K. Mok. Multiprocessor scheduling in a hard real-time environment. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989. 58

[48] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. In *Proceedings of the 12th International Workshop on Parallel and Distributed Real-Time Systems*, April 2004. 8 pages (on CD-ROM). 187

[49] U. Devi and J. Anderson. Schedulable utilization bounds for EPDF fair multiprocessor scheduling. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications, Springer-Verlag Lecture Notes in Computer Science*, pages 261–280, August 2004. 162

[50] U. Devi and J. Anderson. Desynchronized pfair scheduling on multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005. 10 pages (on CD ROM). 38, 332

[51] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, December 2005. 82

[52] U. Devi and J. Anderson. Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006. 10 pages (on CD-ROM). 125, 332

[53] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84, July 2006. 83

[54] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978. 84

[55] J. L. Diaz, D. F. Garcia, K. Kim, C.-G. Lee, L. Bello, J. M. Lopez, S. L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 289–300, December 2002. 50

[56] B. Doytchinov, J. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with Earliest-Deadline-First queue discipline. *Annals of Applied Probability*, 11(2):332–378, 2001. 52

[57] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the USENIX 2005 Annual Technical Conference*, 2005. 333

[58] R. Floyd and R. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975. 112

[59] M. Garey and D. Johnson. *Computers and Intractability : a Guide to the Theory of NP-Completeness*. W. H. Freeman and company, NY, 1979. 29

[60] T. M. Ghazalie and T. P. Baker. Aperiodic servers in deadline scheduling environment. *Real-Time Systems*, 1(9):31–68, 1995. 48

[61] 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. 22, 84, 87, 349

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

[63] M. Hamdaoui and P. Ramanathan. Evaluating dynamic failure probability for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 46(12):1325–1337, December 1997. 44

[64] C.A.R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961. 112

[65] 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. 76

[66] 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. 78

[67] 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. 77

[68] P. Holman and J. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 41–50, July 2003. 77

[69] K. Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of ACM/SIGAPP Symposium on Applied Computing*, pages 796–804, February 1993. 11

[70] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings of the Real-Time Systems Symposium*, pages 304–314, Phoenix, AZ, December 1999. IEEE Computer Society Press. 68, 86

[71] D. Jensen. Time/utility function model of real-time: Worked examples. http://www.real-time.org/casestudies.htm. 53

[72] E. D. Jensen, C. D. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pages 112–122, 1985. 53

[73] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 110–117. IEEE, December 1995. 42

[74] L. Kurk, J. Lehoczky, S. Shreve, and S.-N. Yeung. Earliest-Deadline-First service in heavy-traffic acyclic networks. *Annals of Applied Probability*, 14(3):1306–1352, 2004. 52, 54

[75] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11st IEEE Real-Time Systems Symposium*, pages 201–209. IEEE, December 1990. 49, 149

[76] J. P. Lehoczky. Real-time queueing theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 186–195, December 1996. 52

[77] J. P. Lehoczky, L. Sha, and Y. Ding. Rate-monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989. 18, 49, 149

[78] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 261–270, December 1987. 48

[79] P. Li, B. Ravindran, and E. D. Jensen. Adaptive time-critical resource management using time/utility functions: Past, present, and future. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, volume 2, pages 12–13, 2004. Extended version available at http://www.mitre.org/work/tech_papers/tech_papers_04/. 42, 53

[80] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 193–200, June 2000. 48

[81] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary 37-60*, II:28–31, 1969. 57

[82] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973. 14, 17, 18

[83] J. W. S. Liu, K.-J. Lin, W.-K.Shih, and A. C. Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991. 47

[84] J.W.S. Liu. *Real-Time Systems*, chapter 12, Operating Systems, pages 515–518. Prentice Hall, 2000. 17

[85] J.W.S. Liu. *Real-Time Systems*, chapter 3, A Reference Model of Real-Time Systems, page 42. Prentice Hall, 2000. 48

[86] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004. 22

[87] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 25–34, June 2000. 184

[88] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 211–218, May 2004. 48

[89] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991. 271

[90] 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. 76

[91] A. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-Time Environments*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1983. 14, 17

[92] A. Mok and W. Wang. Window-constrained real-time periodic task scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 15–24, December 2001. 46

[93] D. Oh and T. P. Baker. Utilization bounds for N-processor rate monotone scheduling with static processor assignment. *Real-Time Systems: The International Journal of Time-Critical Computing*, 15:183–192, 1998. 13

[94] A. K. Parekh and R. G. Gallager. 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. 90

[95] K. Ramamritham, J. Stankovic, and P.-F. Shiah. Efficient scheudling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990. 58

[96] S. Ramamurthy and M. Moir. Static-priority periodic scheduling on multiprocessors. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 69–78, December 2000. 75

[97] S. Ramamurthy and M. Moir. Scheduling periodic hard real-time tasks with arbitrary deadlines on multiprocessors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 59–68, December 2002. 75

[98] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999. 45

[99] L. Sha and J. Goodenough. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62, 1990. 146

[100] S. Shankland and M. Kanellos. Intel to elaborate on new multicore processor. http://news.zdnet. co.uk/hardware/chips/0,39020354,39116043,00.htm, September 2003. 336

[101] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989. 48

[102] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, Institut National de Recherche en Informatique et en Automatique, 1996. 149

[103] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996. 48

[104] A. Srinivasan. *Efficient and Flexible Fair Scheduling of Real-Time Tasks on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, December 2003. 65, 71, 81

[105] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002. 22, 66, 67, 68, 73, 79, 80, 81, 90, 165, 167, 168, 189, 192, 359

[106] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 51–59, July 2003. 38, 54, 74, 90, 163, 172, 187, 188, 189, 199

[107] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 77(1):67–80, April 2005. 76, 90, 185

[108] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, September 2006. 90

[109] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002. 84

[110] 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. 54, 334

[111] 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. 10 pages (On CD-ROM). 265, 266, 267

[112] J. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Journal of Real-Time Systems*, 2(4):247–254, March 1990. 2

[113] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28:16–25, June 1995. 58

[114] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299. IEEE, December 1996. 90

[115] T.-S. Tia, D.-Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the 2nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 164–173, May 1995. 49

[116] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 311–320, December 2005. 90

[117] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978. 253

[118] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 239–248. IEEE, December 2000. 46

[119] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems Volume II*, pages 87–91, 1999. 46

[120] D. Zhu, D. Mossé, and R. Melhem. Multiple resource scheduling problem: how much fairness is necessary? In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 142–151, December 2002. 78