REAL-TIME SCHEDULING ON ASYMMETRIC MULTIPROCESSOR PLATFORMS

Kecheng Yang

A dissertation submitted to the faculty at 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
2018

Approved by:

James H. Anderson

Sanjoy K. Baruah

Enrico Bini

Shahriar Nirjon

F. Donelson Smith

ABSTRACT

Kecheng Yang: Real-Time Scheduling on Asymmetric Multiprocessor Platforms
(Under the direction of James H. Anderson)

Real-time scheduling analysis is crucial for time-critical systems, in which provable timing guarantees are more important than observed raw performance. Techniques for real-time scheduling analysis initially targeted uniprocessor platforms but have since evolved to encompass multiprocessor platforms. However, work directed at multiprocessors has largely focused on symmetric platforms, in which every processor is identical. Today, it is common for a multiprocessor to include heterogeneous processing elements, as this offers advantages with respect to size, weight, and power (SWaP) limitations. As a result, realizing modern real-time systems on asymmetric multiprocessor platforms is an inevitable trend. Unfortunately, principles and mechanisms regarding real-time scheduling on such platforms are relatively lacking.

The goal of this dissertation is to enrich such principles and mechanisms, by bridging existing analysis for symmetric multiprocessor platforms to asymmetric ones and by developing new techniques that are unique for asymmetric multiprocessor platforms. The specific contributions are threefold.

First, for a platform consisting of processors that differ with respect to processing speeds only, this dissertation shows that the preemptive global earliest-deadline-first (G-EDF) scheduler is optimal for scheduling soft real-time (SRT) task systems. Furthermore, it shows that semi-partitioned scheduling, which is a hybrid of conventional global and partitioned scheduling approaches, can be applied to optimally schedule both hard real-time (HRT) and SRT task systems.

Second, on platforms that consist of processors with different functionalities, tasks that belong to different functionalities may process the same source data consecutively and therefore have producer/consumer relationships among them, which are represented by directed acyclic graphs (DAGs). End-to-end response-time bounds for such DAGs are derived in this dissertation under a G-EDF-based scheduling approach, and it is shown that such bounds can be improved by a linear-programming-based deadline-setting technique.

Third, processor virtualization can lead a symmetric physical platform to be asymmetric. In fact, for a designated virtual-platform capacity, there exist an infinite number of allocation schemes for virtual processors

and a choice must be made. In this dissertation, a particular asymmetric virtual-processor allocation scheme, called minimum-parallelism (MP) form, is shown to dominate all other schemes including symmetric ones.

ACKNOWLEDGEMENTS

When I joined the Computer Science Department at UNC, I had absolutely zero experience in research. I wanted to stay for five years for a Ph.D., but I was also fine with leaving with a Master's degree. Now, here I am having completed this dissertation. I have enjoyed my life in Chapel Hill as a Ph.D. student so much, and it was the wonderful people I met here who made my life so enjoyable. I would like to take this opportunity to thank many of you.

First and foremost, I would like to thank my advisor Jim Anderson. As a matter of fact, Jim is the creator of my academic career. I do not know why he chose to work with a zero-experience guy like me in the first place, but I have been and will always be grateful for this. I also enjoyed the research discussions with Jim a lot. He always promptly got the point and provided me valuable feedback, and this made our conversations a great pleasure for me. Furthermore, I have learned a great deal about English writing for free in the Computer Science program—this is also because of Jim. I would also like to thank the members of my dissertation committee: Sanjoy Baruah, Enrico Bini, Shahriar Nirjon, and Don Smith. I have also enjoyed and learned a lot from our discussions about and beyond this dissertation.

I also wish thank all of my co-authors: Tanya Amert, Pontus Ekberg, Glenn Elliott, Zhishan Guo, Catherine Nemitz, Nathan Otterness, Luca Santinelli, Sergey Voronov, Shige Wang, and Ming Yang. I am also thankful to many other people I worked with in our group: Bipasa Chattopadhyay, Micaiah Chisholm, Calvin Deutschbein, Jeremy Erickson, Shiwei Fang, Bashima Islam, Tamzeed Islam, Namhoon Kim, Seulki Lee, Rui Liu, Mac Mollison, Sims Osborne, Abhishek Singh, Stephen Tang, and Bryan Ward. I am also grateful to many staff members in the department, most notably Fay Alexander, Robin Brennan, Bridgette Cyr, Jodie Gregoritsch, Adia Ware, and Missy Wood.

Outside of research, soccer games have played a significant role in my life in Chapel Hill. I very much enjoyed playing soccer weekly in the Rivercrabs team and the Newbees team. I could not list the names of all my teammates, because, for many of them, I know their nicknames in the field only. Nevertheless, I would like to thank them all for the numerous soccer afternoons and nights I enjoyed.

Finally, I would like to thank my family, in particular, my parents for their support and patience and Yiqian for her love.

The research in this dissertation was supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, CPS 1446631, and CNS 1563845, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, a grant from General Motors, and a Dissertation Completion Fellowship from the Graduate School at UNC.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMERB	Average Maximum End-to-end Response-time Bound
CE	Computational Element
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DMPR	Deterministic Multiprocessor Periodic Resource
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest-Deadline-First
EDP	Explicit Deadline Periodic
G-EDF	Global Earliest-Deadline-First
G-FL	Global Fair-Lateness
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
HAC	Hardware Accelerator
HOG	Histogram of Oriented Gradients
HRT	Hard Real-Time
I/O	Input/Output
LP	Linear Program
MC	Mixed-Criticality
MP	Minimum-Parallelism
MPR	Multiprocessor Periodic Resource
MSF	Multi Supply Function
PR	Periodic Resource
PSF	Parallel Supply Function
SBF	Supply Bound Function
SRT	Soft Real-Time
SWaP	Size, Weight, and Power

VP	Virtual Processor
VPP	Varying-Period Periodic
WCET	Worst-Case Execution Time

CHAPTER 1: INTRODUCTION

In today's world, computing has become more and more ubiquitous and pervasive. An increasing number of people rely on various computer systems, from lightweight embedded ones to large-scale distributed ones, for a growing range of daily activities. As a result, in many modern systems, computing devices are required to interact with physical processes, some of which are time-critical. Thus, computations in such systems not only need to produce logically sound results but also need to finish in a timely manner. These systems that require both logical and temporal correctness are called *real-time systems*.

To validate the temporal correctness, real-time scheduling-analysis techniques are crucial. After decades of effort, researchers have established a solid foundation for real-time scheduling on uniprocessor platforms, which had been the standard hardware setting for traditional real-time systems for many years. Since the multicore revolution, attention has shifted to supporting multiprocessor platforms. Meanwhile, the multicore revolution is currently undergoing a second wave of innovation in the form of heterogeneous processing elements, which offer advantages with respect to size, weight, and power (SWaP) limitations. As a result, realizing modern real-time systems on asymmetric multiprocessor platforms is an inevitable trend. Unfortunately, principles and mechanisms regarding real-time scheduling on such platforms are relatively lacking.

The goal of this dissertation is to enrich such principles and mechanisms, by bridging existing analysis for symmetric multiprocessor platforms to asymmetric ones and by developing new techniques that are unique for asymmetric multiprocessor platforms.

We begin this chapter with an introduction to real-time systems. We also present an overview of a category of asymmetric platforms that are considered in this dissertation. We then state the thesis, summarize the contributions, and outline the remaining chapters of this dissertation.

1.1 Real-Time Systems

A real-time system requires the validation of both *logical* and *temporal* correctness. Logical correctness, which needs to be verified for almost all computing systems including non-real-time ones, requires the system

to always produce the right outputs, whereas temporal correctness further requires the system to do so at the right time. Since logical correctness is a general problem faced by virtually any system, the particular interest of real-time systems research mostly lies in temporal correctness.

1.1.1 Workload Model and Temporal Correctness Criterion

In a real-time system, the timing of the completion of workloads is not only a matter of efficiency but also a matter of correctness. It is therefore critical to validate the temporal properties of a real-time system before it runs, or even before it is built. Thus, properly modeling the workloads is the first step towards establishing temporal correctness in a real-time systems.

Since the seminal paper by Liu and Layland (1973), the classic *sporadic task model* has been widely received as a fundamental real-time workload model. A sporadic task τ_i releases a (potentially infinite) set of jobs, where consecutive job releases are separated by at least T_i time units and each job has a *worst-case execution requirement* C_i, which is defined by its worst-case execution time on a unit-speed processor. If the separation between *every* consecutive jobs happens to be *exactly* T_i time units, then τ_i is also called a *periodic* task. For this reason, T_i is called the *period* of τ_i, regardless of whether τ_i is periodic or sporadic. The *utilization* of τ_i is defined as $u_i = C_i/T_i$, which indicates the amount of computing resources τ_i will request in the long term.

Each job of τ_i has an *absolute deadline*, or simply deadline, D_i time units after its release time, where D_i is called the *relative deadline* of τ_i. Deadlines are called *implicit* if $D_i = T_i$. The j^{th} job of τ_i is denoted by $\tau_{i,j}$, and its release time, finish time, and deadline are denoted by $r_{i,j}$, $f_{i,j}$, and $d_{i,j}$, respectively. The *response time* of job $\tau_{i,j}$ is defined by $f_{i,j} - r_{i,j}$, *i.e.*, the time duration from its release time to its finish time. Furthermore, if $\tau_{i,j}$ misses its deadline at $d_{i,j}$, then the difference between its finish time and its deadline is called its *tardiness*; on the other hand, if $\tau_{i,j}$ meets its deadline, then its tardiness is defined to be zero. That is, the tardiness of job $\tau_{i,j}$ is define by $\max\{0, f_{i,j} - d_{i,j}\}$.

In a hard real-time (HRT) system, its temporal correctness requires that every job must be guaranteed to complete by its deadline, *i.e.*, the tardiness of every job must be zero. In contrast, in a soft real-time (SRT)[1] system, its temporal correctness can be validated as long as the tardiness of every job can be upper-bounded by some constant.

[1] In this dissertation, we adopt the tardiness-based definition of SRT systems. For other definitions of SRT systems, please see Erickson's dissertation (Erickson, 2014) for a review.

In both HRT and SRT systems, the response time for every job must be upper-bounded for the corresponding temporal correctness to be established. In HRT systems, such as flight control systems, deadlines represent very specific response-time requirements for the jobs and such requirement are explicit to the system designers. Therefore, response-time bounds equal to corresponding relative deadlines are required for temporal correctness. In SRT systems, such as real-time multimedia streaming systems, deadlines may be just a notion of urgency provisioned by the system designers, where some response-time guarantees that are not specified explicitly but are derived from analysis may be acceptable. For example, a response time of 1 ms, 2 ms, or 100 ms may not be noticeable in a live basketball game, and even 2,000 ms or 5,000 ms could be sufficiently good for most audiences. Some constant response-time bounds, *e.g.*, a relative deadline plus a constant tardiness bound, are sufficient to guarantee that the response times of jobs will not unboundedly grow, so that the live streaming will not be choppy, given adequate buffering at the very beginning.

1.1.2 Schedulability, Feasibility, and Optimality

When multiple real-time tasks share a common processing platform for their execution, competition among tasks for processing resources is inevitable. When such competition occurs, the "judge" who dictates the resource allocation for the tasks is called a *scheduler*, and the algorithm a scheduler runs is called a *scheduling algorithm*. The term *schedulable* (*feasible*, respecttively) is defined for a real-time system with respect to a particular scheduling algorithm (some scheduling algorithm, respectively), while the term *optimal* is defined for a scheduling algorithm with respect to all *feasible* real-time systems.

Definition 1.1. (schedulable) A real-time system is called *HRT-schedulable* (*SRT-schedulable*, respectively) under a scheduling algorithm $\mathcal{A}$ if and only if the tardiness of every job in that system is guaranteed to be zero (upper-bounded by a constant, respectively) under the scheduling algorithm $\mathcal{A}$.

Definition 1.2. (feasible) A real-time system is called *HRT-feasible* (*SRT-feasible*, respectively) if and only if this system is HRT-schedulable (SRT-schedulable, respectively) under some scheduling algorithm.

Definition 1.3. (optimal) A scheduling algorithm is called *HRT-optimal* (*SRT-optimal*, respectively) if and only if all HRT-feasible (SRT-feasible, respectively) systems are HRT-schedulable (SRT-schedulable, respectively) under this scheduling algorithm.

The concept of optimality is also often defined with respect to a subset of real-time systems. In this case, feasibility should be interpreted accordingly within that subset of real-time systems as well. For example,[2]

"Scheduling algorithm $\mathcal{A}$ is optimal *on uniprocessors*."

should be interpreted as

"All feasible systems *on uniprocessors* are schedulable under $\mathcal{A}$."

where "all feasible systems *on uniprocessors*" should be interpreted as "all real-time systems *on uniprocessors* that are schedulable under some algorithm."

1.2 Asymmetric Multiprocessor Platforms

The aforementioned classic sporadic task model and its extensions have been widely studied on uniprocessor platforms since it was proposed by Liu and Layland (1973). Since the multicore revolution, it has mostly been studied with respect to *symmetric* multiprocessor platforms, where every processor is viewed as identical. In contrast, there is a relative lack of similar work addressing asymmetric multiprocessor platforms, where processors may differ from each other. Due to the additional complexity that may arise on such platforms, asymmetric multiprocessor platforms must be modeled more carefully.

The following is a taxonomy of multiprocessors from symmetric to asymmetric, adopted from Pinedo (1995) and Funk (2004).

- **Identical.** Every job is executed on any processor at the same speed, which is usually normalized to be 1.0 for simplicity.
- **Uniform.** Different processors may have different speeds, but on a given processor, every job is executed at the same speed. The speed of processor p is denoted s_p.
- **Unrelated.** The execution speed of a job depends on both the processor on which it is executed and the task to which it belongs, *i.e.*, a given processor may execute jobs of different tasks at different speeds. The execution speed of task τ_i on processor p is denoted $s_{p,i}$.

The uniform multiprocessor model might be the most straightforward step from a symmetric multiprocessor platform to an asymmetric one. It allows each processor on the platform to have its own *speed*; workloads

[2]In this example, "HRT" and "SRT" are omitted for simplicity. They can be added in the same fashion as the three definitions above.

performed on each processor progress proportionally, or *uniformly*, to the corresponding speed. This is indeed a rather idealistic platform model, because different tasks may often scale differently on two given different-speed processors, *i.e.*, the speeds of processors can be difficult to determine in practice. Nonetheless, this model is still of interest because of its simplicity. Theoretical results involving this model can be viewed as a baseline for moving forward to more complicated system models and might reveal fundamental intuitions on the differences between symmetric and asymmetric multiprocessor platforms.

On the other hand, the unrelated multiprocessor model might be a more expressive model, which may be able to represent the case in which processors may not only differ with respect to processing speeds but also have different *functionalities*. However, for a particular job, each processor is still characterized by a single-value speed, although it could be different values for different jobs. This results in the implication of a uniform execution within a single job. That is, executing the first half of a job on processor p and executing the second half on processor q is the same as executing the first half of a job on processor q and executing the second half on processor p. However, this might not be true in practice if processors p and q have different functionalities. For example, if processor p is "better" at doing the computations in the first half while processor q is "better" for those in the second half, then the former allocation may have a significantly shorter worst-case execution time than the latter. Thus, in many actual systems, migrations among processors of different functionalities are not allowed, at least at the job level. In addition, it is often the case that only a few different functionalities exist while multiple processors may be of the same functionality. With these constraints, an unrelated multiprocessor could be able to model processors of different functionalities.

Moreover, even an identical multiprocessor platform may not necessarily imply the absence of asymmetry, if hardware *virtualization* is considered. Compositional real-time systems have been proposed in the literature to support open systems (Deng and Liu, 1997), where separate software components execute together on a common hardware platform while each component can be developed, analyzed, and certified independently. In such systems, each component has the "illusion" of executing on a dedicated *virtual* platform, and it should be possible to validate the temporal correctness of each component independently. Therefore, a virtual platform needs to be specified, usually by virtual processors, which are used to characterize partially available physical processors. Since different characteristics of the availability of a certain physical processor may result in different virtual processors, a virtual multiprocessor platform can be in fact asymmetric, even if the underlying physical platform consists of only identical processors.

1.3 Thesis Statement

Due to different sources of asymmetry, the modeling of an asymmetric multiprocessor platform is significantly more complicated than that of simpler traditional platforms, and therefore designing real-time systems on such platforms is also much more challenging. In this dissertation, we consider a set of asymmetric multiprocessor models, each of which addresses a particular source of asymmetry, and we develop and analyze scheduling algorithms for real-time systems design under each model. This leads to the following thesis statement:

> *Multiprocessor platforms can be asymmetric due to differing processor speeds, differing processor functionalities, or virtualization. On platforms with different-speed processors, optimal SRT scheduling does not require radically new scheduling techniques, but rather can be done by simply adapting certain techniques that are known to be SRT-optimal on platforms with identical processors. On platforms comprised of processors with different functionalities, dataflow computations can be more effectively supported if prioritization schemes are used that holistically account for end-to-end data-processing objectives. On asymmetric virtual multiprocessor platforms, there exists a single virtual-processor allocation scheme that dominates all other schemes.*

1.4 Contributions

The above thesis is supported by the following contributions, summarized by the three aspects of asymmetry considered in this dissertation.

1.4.1 Asymmetric Platforms Due to Differing Processor Speeds

For a system consisting of processors that differ with respect to processing speeds only, we model it as a uniform multiprocessor. Since the identical multiprocessor model is a special case of the uniform multiprocessor model, where the speed of every processor happens to be 1.0, we begin with the global earliest-deadline-first (G-EDF) scheduling algorithm, which has been widely studied on identical multiprocessors.

In Chapter 3, we examine the SRT-optimality of G-EDF on uniform multiprocessors. Devi and Anderson (2008) have established the SRT-optimality of G-EDF on identical multiprocessors, but the problem of

whether this optimality result also holds on uniform multiprocessors has been open since then. As noted in Section 1.1.2, when optimality is discussed for a certain set of systems, feasibility must be considered accordingly. Therefore, we first establish a SRT-feasibility condition for uniform platforms as follows:

> A set of n sporadic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is SRT-feasible on m uniform processors with speeds $\{s_1, s_2, \ldots, s_m\}$ if and only if
>
> $$U_n \leq S_m, \text{ and} \tag{1.1}$$
>
> $$U_k \leq S_k, \text{ for } k = 1, 2, \ldots, m-1, \tag{1.2}$$
>
> where U_k denote the sum of the k largest utilizations of tasks in τ and S_k denote the sum of the k largest processor speeds.

This in fact matches the HRT-feasibility condition for *implicit-deadline periodic* tasks on uniform platforms (Funk et al., 2001). According to Devi and Anderson (2008), both preemptive and non-preemptive G-EDF scheduling algorithms are SRT-optimal on identical multiprocessors. Therefore, we also investigate G-EDF on uniform multiprocessors with respect to these two variants. By constructing a counterexample in Chapter 3, we show that the non-preemptive G-EDF scheduling algorithm is not SRT-optimal on uniform platforms. In contrast, the preemptive G-EDF scheduling algorithm remains SRT-optimal on uniform platforms. A rather long and complicated proof will be presented in Chapter 3 to support this optimality result.

In Chapter 4, we shift our focus from G-EDF, which follows a *global* scheduling approach, to a different scheduling approach, called *semi-partitioned* scheduling. Traditionally, a scheduling algorithm usually follows either a global or a partitioned approach. Global scheduling approach allows *any* migrations, while partitioned scheduling approach allows *no* migration at all. One classic hybrid of these two approaches is *clustered* scheduling, where only migrations within a subset of *processors*, or a cluster, are allowed. *Semi-partitioned* scheduling is another hybrid, where only migrations of a subset of *tasks*, called *migrating tasks*, are allowed and all other tasks, called *fixed tasks*, cannot migrate. In Chapter 4, we will design and analyze two semi-partitioned scheduling algorithms, namely EDF-sh (earliest-deadline-first-based semi-partitioned scheduler for uniform heterogeneous multiprocessors) and EDF-tu (earliest-deadline-first-based tunable scheduler for uniform platforms), for uniform multiprocessors. EDF-sh is neither HRT- nor SRT-optimal, but it restricts task migrations to occur at job boundaries only, *i.e.*, no job migrates even if it is from a migrating

task. EDF-tu includes a tunable parameter, so that its HRT- or SRT-optimality and its tardiness bounds can be tuned by adjusting this parameter at the expense of potentially increased runtime overheads.

In Chapter 5, we re-visit the sporadic task model we consider in the prior two chapters. In the conventional sporadic task model, each task is considered as a sequential schedulable entity that is often implemented by a single thread in a real system. Therefore, the jobs of a single sporadic task must execute in sequence. That is, a job cannot commence execution until all its predecessors finish, even if available processors exist. This intra-task precedence constraint might jeopardize the feasibility of a system where such a constraint is in fact unnecessary. For example, when a video is processed frame-by-frame independently, consecutive jobs (*i.e.*, processing consecutive frames) can execute in parallel, if sufficient processors are available on a multiprocessor platform. To remove this intra-task precedence constraint when it is unnecessary, in Chapter 5 we introduce a new task model, called *npc-sporadic (non-precedence-constraint sporadic)* task model, in which multiple jobs of the same task may execute in parallel (while each individual job remains sequential). With the npc-sporadic model, we will show that Equation (1.2) is not necessary for SRT-feasibility on uniform platforms anymore. Furthermore, the non-preemptive G-EDF scheduling algorithm becomes SRT-optimal again, and the preemptive G-EDF scheduling algorithm remains SRT-optimal with better tardiness bounds.

1.4.2 Asymmetric Platforms Due to Differing Processor Functionalities

In Chapter 6, we consider platforms that consist of processors with different functionalities. As mentioned in Section 1.2, migrations on such platforms may be subtle, so that even the unrelated multiprocessor model might also fail to characterize these platforms. Specifically speaking, on an unrelated multiprocessor platform, a *slowest* processor can be identified for each task, and that task's worst-case execution time (WCET) can be correspondingly defined with respect to its execution on that processor. By then determining WCETs for the task on other processors, appropriate per-processor speed parameters can be defined for the task. Such WCET values and speed parameters are valid, provided the task never migrates. However, if migrations are allowed, then it can be the case that the highest WCET specified for the task—that obtained by considering its slowest processor—is not actually the largest possible WCET for the task, even if migration costs are assumed to be negligible. That is, *migrations can cause anomalies with respect to execution-time and speed assumptions*. To see this, consider a task τ_1, which executes the function `f()` in Figure 1.1. Suppose that τ_1 has a WCET of 100 ms when executing entirely on a processor p, and a WCET of 200 ms when executing entirely on a processor q, where processors p and q are two processors of different functionalities. Then, the

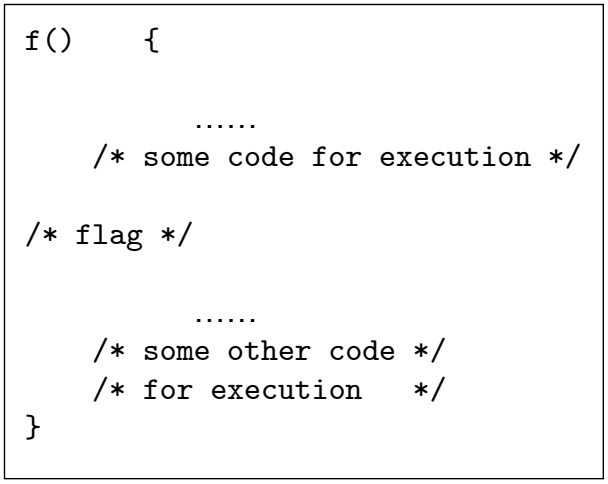


Figure 1.1: An example program structure.

following anomaly is possible: suppose that the WCET for executing from the beginning of `f()` to the line `flag` is 90 ms on processor p and 50 ms on processor q, and the WCET for executing from the line `flag` to the end of `f()` is 10 ms on processor p and 150 ms on processor q. In such a situation, if a job of τ_1 first executes on processor p until the line `flag` and then migrates to processor q to finish its execution, then it could execute for 240 ms, which is greater than any of the WCETs defined for τ_1 when it executes entirely on a single processor type.

The above anomaly occurs because the two processors of different functionalities may "favor" different kinds of computations, which could exist in different pieces of code, even in the same task. Furthermore, the line of `flag` could potentially be any line in the code of the task, and this makes the WCETs and a set of valid speeds for the unrelated multiprocessor model extremely difficult to obtain. Therefore, we consider the situation where processors are pooled by their functionalities and tasks are assigned to a particular pool consisting of a set of identical processors. In this case, inter-pool migrations are not allowed so that the described anomaly is eliminated, but migrations among processors in the same pool may be allowed. Tasks are scheduled within each pool, which is a symmetric platform; if the tasks are independent, this is a rather well-studied problem. However, if producer/consumer relationships may exist among tasks, the isolated pools would be re-connected by such relationships between tasks and a more holistic analysis on the entire task set on processors of different functionalities is needed. In Chapter 6, we deal with such systems where the producer/consumer relationships among tasks may be represented by *directed acyclic graphs* (DAGs). By applying G-EDF-based scheduling, an *end-to-end response-time bound* can be derived for each DAG.

Moreover, such bounds can be further improved by leveraging linear-programing (LP)-based techniques to tune the deadline setting for each individual task.

1.4.3 Asymmetric Platforms Due to Virtualization

In Chapter 7, we consider a potentially asymmetric virtual multiprocessor platform that is implemented on a symmetric physical multiprocessor platform. Virtual platforms, which are usually described by virtual processors, have been proposed to enable the design, analysis, and validation of sub-systems on a common shared physical platform.

In early work in this direction pertaining to uniprocessor platforms, Shin and Lee (2003) proposed a virtual processor (VP) model called the *periodic resource (PR)* model, which allows the considerable body of work on periodic and/or sporadic task scheduling to be exploited in reasoning about the allocation of processor time to components. In the PR model, a VP is specified by the parameters (Π, Θ), with the interpretation that Θ time units of processor time is guaranteed to the supported component every Π time units. While this simple model sufficed in the uniprocessor case, it is inadequate in the multiprocessor case, because the important issue of *parallelism* is ignored. To deal with this issue, Shin et al. (2008) proposed extending the PR model by adding an additional parameter. Specifically, under their *multiprocessor periodic resource (MPR)* model, the supply allocated to a component is specified by (Π, Θ, m'), with the interpretation that Θ time units of processor time is guaranteed to the component every Π time units with at most m' VPs providing allocation in parallel. That is, the new parameter m' specifies the *maximum degree of parallelism*. In the MPR model, all VPs allocated to a component are required to have a common period Π that is strictly synchronized.

A key characteristic of the MPR model is its flexibility. For example, consider a component that is to be allocated 80% of the capacity of a quad-core machine. The supply interface for that component could be defined as $(100, 320, 4)$, meaning that every 100 time units, the component receives 320 units of processing time on up to four processors. Such a specification does not indicate the precise manner in which processing time is allocated. For example, the component could be allocated 80% of the capacity of each processor, or 100% of three processors and 20% of the fourth, among other choices. Which choice is best?

In the example above, the second-listed choice is known as *minimum-parallelism (MP) form*. Under MP form, each component is allocated at most one partially available processor, with all other processors allocated to it being fully available. MP form was first proposed by Leontyev and Anderson (2009), who

have shown that MP form is the best for supporting SRT tasks. We extend this result in Chapter 7 by showing that MP form is in fact the best for HRT tasks as well, provided that each VP is modeled by the PR model and the virtual platform is characterized by *parallel supply functions* (Bini et al., 2009a).

1.5 Organization

In Chapter 2, we review general background and related prior work; each subsequent chapter is self-contained with notations and definitions that are specific to it. Focusing on the uniform multiprocessor model, we then prove both the negative and positive results regarding the SRT-optimality of G-EDF in Chapter 3, present two semi-partitioned scheduling algorithms in Chapter 4, and discuss the npc-sporadic task model in Chapter 5. Next, in Chapter 6, we provide techniques to establish and improve end-to-end response-time bounds for DAG-based task systems on processors of different functionalities. Afterwards, we establish the dominance of MP form on virtual multiprocessor platforms in Chapter 7. Finally, we conclude in Chapter 8.

CHAPTER 2: BACKGROUND

In this chapter, we provide needed background for this dissertation by surveying related prior work, and by highlighting the contributions of this dissertation in the context of the existing literature.

2.1 HRT-Feasibility and HRT EDF Scheduling on Uniform Multiprocessors

Under the uniform multiprocessor model, a platform π has m processors, where processor p is identified by its speed s_p ($1 \leq p \leq m$, $s_p \in \mathbb{R}$). In this dissertation, when considering a uniform multiprocessor, we index processors in non-increasing-speed order, *i.e.*, $\pi = \{s_1, s_2, \cdots, s_m\}$, where $s_p \geq s_{p+1}$ for $1 \leq p \leq m-1$; we also index tasks in non-increasing-utilization order, *i.e.*, $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where $u_i \geq u_{i+1}$ for $1 \leq i \leq n-1$.

By leveraging the Level Algorithm (Horvath et al., 1977), Funk et al. (2001) showed that an *implicit-deadline periodic* task system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is HRT-feasible if and only if

$$U_n \leq S_m, \text{ and} \tag{2.1}$$

$$U_k \leq S_k, \text{ for } k = 1, 2, \ldots, m-1, \tag{2.2}$$

where $U_k = \sum_{i=1}^{k} u_i$ and $S_k = \sum_{i=1}^{k} s_i$.

Subsequently, Funk and her colleagues developed several EDF-based scheduling algorithms that support HRT tasks on uniform multiprocessors (Funk, 2004).

In the rest of this section, we review the Level Algorithm and those EDF-based HRT algorithms by Funk (2004). A detailed understanding of the Level Algorithm is needed because it will be used as a subroutine in Chapter 4.

2.1.1 Level Algorithm

The Level Algorithm was proposed by Horvath et al. (1977) for scheduling a set of non-real-time jobs on a uniform multiprocessor with the goal of minimizing *makespan*, *i.e.*, the time required for finishing all jobs. A job's *level* is defined by its remaining execution time. The greater a job's level, the faster the processor on

which it is scheduled, and all jobs that attain the same level are thereafter *jointly executed*, equally sharing the processors on which they are scheduled. The following example illustrates the Level Algorithm.

Example 2.1. Consider using the Level Algorithm to schedule four jobs, with initial execution requirements $J_1 = 12$, $J_2 = 12$, $J_3 = 8.5$, and $J_4 = 7.5$, on a uniform platform $\pi = \{s_1 = 4, s_2 = 3, s_3 = 2, s_4 = 1\}$. J_1 and J_2 have the same execution cost, or level, so they are jointly executed from the beginning; J_3 and J_4 attain the same level at time 1, so they are jointly executed after time 1. At time 2, all jobs attain the same level, and hence all jobs are jointly executed afterward. Figure 2.1 shows the resulting schedule generated by the Level Algorithm for this example. Figure 2.2 shows the actual schedule for "jointly executing." Figure 2.3 shows the actual schedule for the system. Figure 2.4 shows a slight variation of the actual schedule that leverages the fact that, when jobs start to jointly execute, we can make every processor involved in this joint execution start with its currently executing job to reduce unnecessary preemptions and migrations. ◊

Theorem 2.1 (Theorem 1 in (Horvath et al., 1977))**.** *Let $\mathcal{J} = \{J_1, J_2, \cdots, \mathcal{J}_n\}$ denote a set of independent non-real-time jobs to be scheduled on an m-processor uniform multiprocessor π. Let X_i denote the sum of the i largest execution requirements in $\mathcal{J}$. Then the Level Algorithm constructs a minimum makespan, which is given by*

$$\max\left(\max_{1\le i\le m-1}\left\{\frac{X_i}{S_i}\right\}, \frac{X_n}{S_m}\right).$$

This is very similar to the feasibility condition given by (2.1) and (2.2), because that feasibility condition was, in fact, derived from the Level Algorithm (Funk et al., 2001).

2.1.2 HRT EDF Scheduling on Uniform Multiprocessors

Most prior work regarding EDF scheduling for HRT systems on uniform multiprocessors is by Funk and her colleagues. This work is summarized in detail in Funk's dissertation (Funk, 2004).

Funk and Baruah (2003) considered the HRT-schedulability problem for EDF with full migration (f-EDF) on a uniform multiprocessor. In earlier work, Baruah et al. (2003) showed that f-EDF scheduling on uniform multiprocessors is *robust* with respect to the processing platform. That is, replacing a uniform multiprocessor by a more *powerful* one does not compromise HRT-schedulability under f-EDF, where a uniform multiprocessor π_1 is said to be more powerful than another one π_2 if m_1—the number of processors on π_1—is at least m_2—the number of processors on π_2—and the i^{th} fastest processor on π_1 is no slower than i^{th} fastest processor on π_2 for $1 \le i \le m_2$. In light of this robustness result, Funk and Baruah (2003)

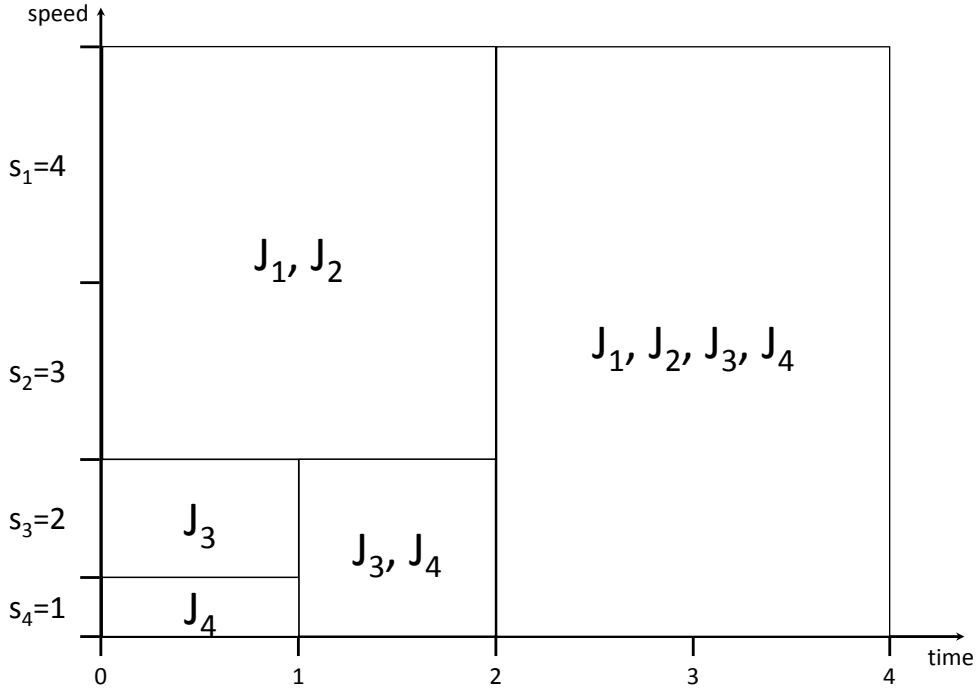


Figure 2.1: "Jointly executing" schedule for Example 2.1 generated by the Level Algorithm.

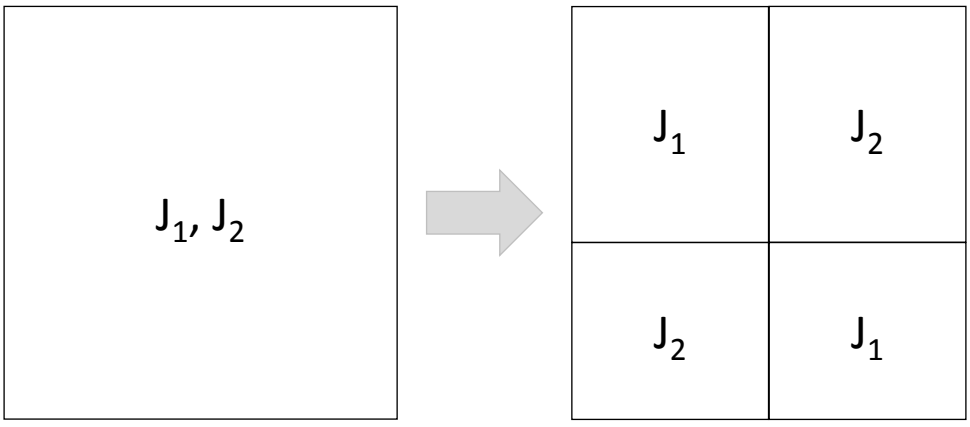


Figure 2.2: The actual schedule for "jointly executing" J_1 and J_2.

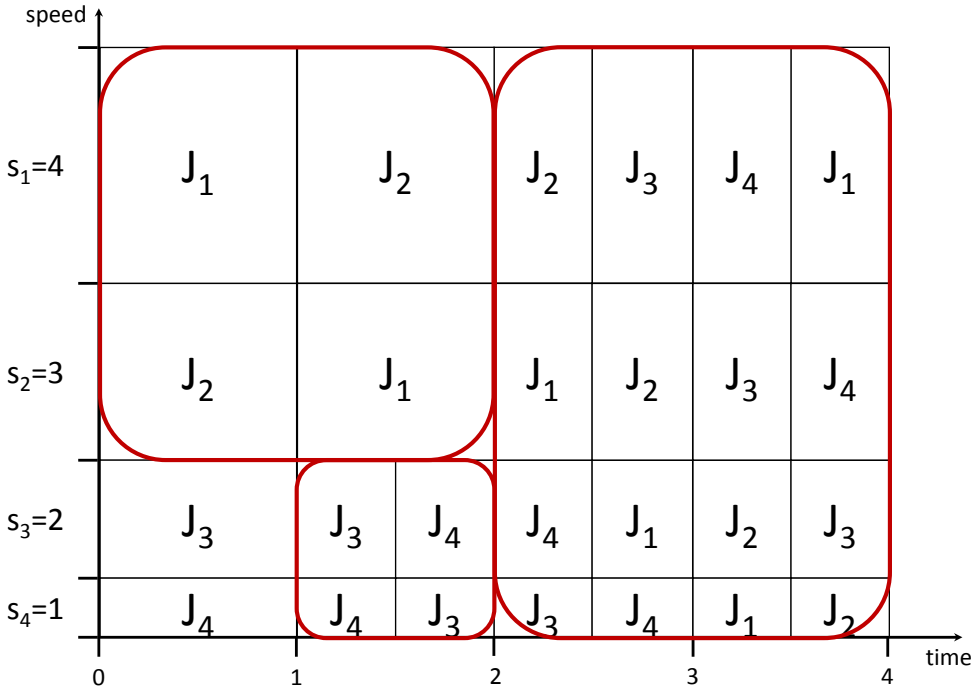


Figure 2.3: Actual schedule for Example 2.1 generated by the Level Algorithm.

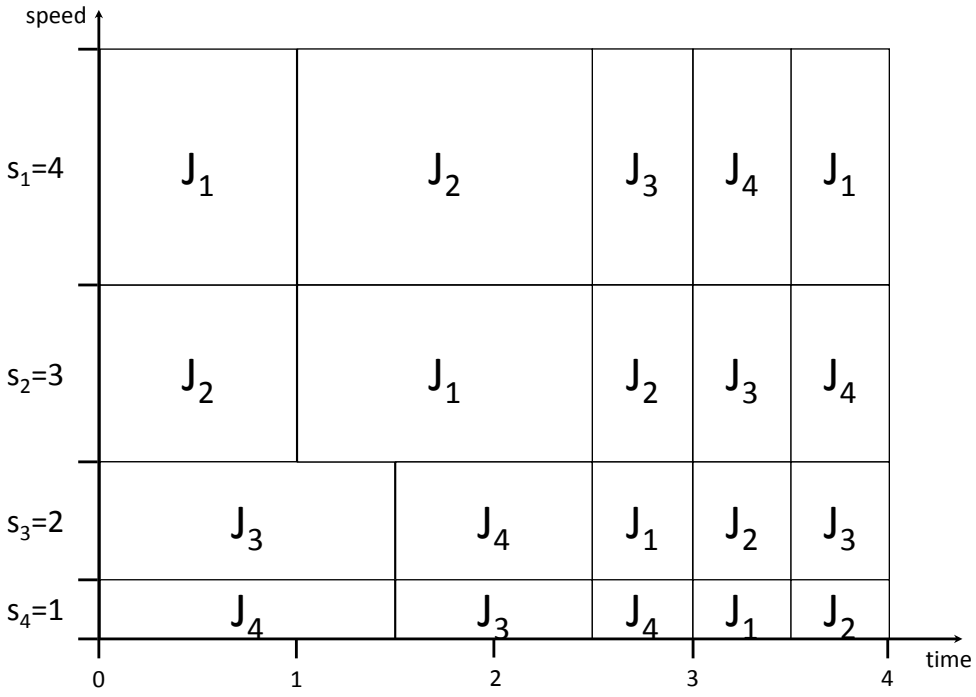


Figure 2.4: Final schedule for Example 2.1 generated by the Level Algorithm.

reduced the f-EDF-HRT-schedulability problem on an arbitrary uniform multiprocessor platform π to an HRT-feasibility problem on a less powerful platform π'. Therefore, an HRT-schedulability test follows by leveraging existing HRT-feasibility results.

Funk and Baruah (2005a) then shifted their attention to EDF scheduling with restricted migration (r-EDF), which requires migrations to occur at job boundaries only. They presented an r-EDF scheduler for uniform multiprocessors by extending results by Baruah and Carpenter (2005) pertaining to identical multiprocessors. The r-EDF scheduler proposed by Funk and Baruah (2005a) migrates tasks at job boundaries only by assigning each job to a particular processor at its release and then using a uniprocessor EDF scheduler on each individual processor. They provided a utilization-bound-based HRT-schedulability test for the r-EDF scheduler on a uniform multiprocessor, and presented two techniques to improve schedulability, namely "semi-partitioning" and "virtual processors."[1]

Funk and Baruah (2005b) also investigated partitioned EDF (p-EDF) scheduling on a uniform multiprocessor. They focused on p-EDF scheduling with the any-fit-decreasing (AFD) task-assignment heuristic. They presented a framework to approximate the utilization bound for applying AFD-EDF on an arbitrary uniform multiprocessor. This bound depends only on the maximum per-task utilization and the total system utilization, and any task system satisfying this bound can be successfully partitioned by the AFD heuristic and therefore is HRT-schedulable under AFD-EDF.

Contributions of this dissertation. In Chapter 3, we will extend the consideration of HRT-feasiblity on a uniform multiprocessor to incorporate SRT systems as well. Furthermore, the problem of whether G-EDF is SRT-optimal will be discussed as well. In Chapter 4, we will present two EDF-based semi-partitioned scheduling algorithms for uniform multiprocessors. In particular, one of them restricts migrations to occur on job boundaries only, and the other one is SRT-optimal and can be HRT-optimal under certain settings.

2.2 Tardiness Bounds under G-EDF Scheduling

As mentioned in Chapter 1, we have adopted the tardiness-based definition of SRT systems in this dissertation. In this section, we review related prior work on tardiness bounds. Most of this work focuses on identical multiprocessors.

[1] Both terms as used by us in this dissertation have a different meaning from (Funk and Baruah, 2005a). We refer interested readers to (Funk and Baruah, 2005a) for details concerning the usage of these two techniques in their context.

Devi and Anderson (2008) proved tardiness bounds for implicit-deadline sporadic task systems under G-EDF scheduling on an identical multiprocessor. Their bounds apply to both preemptive and non-preemptive G-EDF for any (HRT- or SRT-) feasible system. Thus, the SRT-optimality of G-EDF was first established by them. Subsequently, G-EDF tardiness bounds were improved by Erickson et al. (2010a) by introducing the concept of *compliant vectors*, and extended by Erickson et al. (2010b) to apply to sporadic tasks with arbitrary deadlines. Recently, Valente (2016) proposed an alternative approach to compute tighter tardiness bounds for preemptive G-EDF at the cost of higher complexity. This work was then improved by Leoncini et al. (2017) to compute such tardiness bounds more efficiently.

Meanwhile, beyond G-EDF scheduling, Leontyev and Anderson (2007b) derived tardiness bounds for sporadic task systems under first-in-first-out (FIFO) scheduling on an identical multiprocessor. Leontyev and Anderson (2010) then extended both tardiness bounds for G-EDF and FIFO, resulting in a framework for deriving such tardiness bounds under a class of global schedulers called *window-constrained schedulers*, which include the G-EDF scheduler, the FIFO scheduler, and all other *G-EDF-like* schedulers. Under G-EDF-like scheduling, each job has a *priority point*: the earlier the priority point, the higher the priority; every job of the same task has the same constant difference between its priority point and release time. The G-EDF and FIFO schedulers are both G-EDF-like schedulers: under G-EDF, the priority point of a job is at its absolute deadline; under FIFO, the priority point of a job is at its release time. This work raised the question: which G-EDF-like scheduler provides the best tardiness bounds? Erickson et al. (2014) answered this question by presenting the global fair-lateness (G-FL) scheduler.

The only prior work that addressed G-EDF-related tardiness bounds on asymmetric multiprocessors was by Tong and Liu (2016). They derived tardiness bounds under a G-EDF variant on a multiprocessor platform consisting of processors of different speeds. However, they imposed certain utilization restrictions on the task system to which their tardiness bounds apply and therefore did not establish any SRT-optimality result. We will discuss such limitations in more detail in Chapter 3.

Contributions of this dissertation. In Chapter 3, we will focus on extending the SRT-optimality results for G-EDF on identical multiprocessors to uniform multiprocessors. Assuming a uniform multiprocessor platform, we will disprove the SRT-optimality of non-preemptive G-EDF by giving a counterexample, and we will prove the SRT-optimality of preemptive G-EDF by deriving a tardiness bound. This work extends the literature on SRT-optimality from identical multiprocessors to uniform multiprocessors. Moreover, in

contrast to (Tong and Liu, 2016), our work focuses on optimality results and therefore imposes no utilization restrictions other than those required for feasibility.

2.3 Semi-Partitioned Scheduling Algorithms

In Chapter 4, we will present two *semi-partitioned* scheduling algorithms for uniform multiprocessors. In this section, we review existing work on semi-partitioned scheduling prior to this dissertation. Most of this work targeted identical multiprocessors.

Semi-partitioned scheduling. Traditionally, a multiprocessor scheduling algorithm follows either a global or a partitioned approach. The former allows any migrations while the latter allows no migration at all. As a hybrid, semi-partitioned scheduling allows only a subset of tasks to migrate. A semi-partitioned scheduling algorithm usually has two phases: an *assignment phase* and an *execution phase*. During the assignment phase, each task is allocated a non-zero *share* on certain processors such that the total allocated share on each processor does not exceed the processor's capacity and the total allocated share of a task matches its utilization. If a task has non-zero shares on only one (respectively, multiple) processor(s), then it is a *fixed* (respectively, *migrating*) task. During the execution phase, the scheduler needs to schedule tasks according to their allocated shares while providing timing guarantees for each task.

Semi-partitioned scheduling was proposed by Anderson et al. (2005) by presenting EDF-fm. Subsequently, a number of semi-partitioned scheduling algorithms for identical multiprocessors were proposed: Andersson and Tovar (2006) proposed EKG, Kato and Yamasaki (2007) proposed Ehd2-SIP, Kato and Yamasaki (2008) proposed EDDP, Andersson et al. (2008) proposed EDF-SS, Bletsas and Andersson (2009) proposed a concept of "notional processors," Kato and Yamasaki (2009) proposed EDF-WM, Guan et al. (2010) proposed SPA1 and SPA2, Bletsas and Andersson (2011) proposed NPS-F, Burns et al. (2012) proposed a "C=D" task-splitting scheme, Bhatti et al. (2012) proposed 2L-HiSA, Fan and Quan (2012) proposed HSP-light and HSP, Sousa et al. (2013) proposed Carousel-EDF, Anderson et al. (2016) proposed EDF-os, and Patterson and Chantem (2016) proposed EDF-hv. In addition, empirical studies for semi-partitioned scheduling were conducted by Bastoni et al. (2011) and Brandenburg and Gül (2016).

Beyond identical multiprocessors, techniques from semi-partitioned scheduling were used to design EDF-ms (Leontyev and Anderson, 2007a), which is able to support a multiprocessor platform consisting of processors of different speeds. EDF-ms in fact is not a semi-partitioned scheduler; instead, it divides

processors into groups by their speeds and assigns jobs to groups by a "semi-partitioned" approach similar to EDF-fm at the group level; global scheduling is employed within each group.

In the following, we will review EDF-fm, EDF-os, and EDF-ms in more detail, as they will be directly referred to in Chapter 4.

2.3.1 EDF-fm

EDF-fm (Anderson et al., 2005) is an EDF-based semi-partitioned scheduling algorithm for sporadic task systems on an identical multiprocessor. It has an assignment phase and an execution phase. In the assignment phase, processors and tasks are considered in turn. Each considered task is assigned to the currently considered processor until its capacity is exhausted. In this case, the remaining share, if any, of this task that exceeds the capacity of the currently considered processor will be assigned to next processor, which will then become the "currently considered processor" for subsequent unassigned tasks. Thus, there are at most two migrating tasks on each processor—the first assigned one and the last assigned one—and each migrating task has non-zero shares on exactly two processors.

In the execution phase, all jobs of a fixed task are dispatched to the processor to which that task was assigned, whereas the jobs of a migrating task need to be dispatched to processors by a more sophisticated mechanism. The goal of this job-dispatching mechanism is to limit migrations to occur on job boundaries only while ensuring that the allocated shares on the two processors to which a migrating task was assigned are maintained in the long run. To this end, EDF-fm dispatches an appropriate *fraction* of jobs of a migrating task to each of the two processors to which it was assigned, by leveraging results from Pfair Scheduling (Baruah et al., 1996). This job-dispatching mechanism provides the following property, where $\varphi_{i,p}$ denotes the fraction of jobs of task τ_i to execute on processor p.

Property 2.1. *For the first z jobs of task τ_i, at least $\lfloor \varphi_{i,p} \cdot z \rfloor$ and at most $\lceil \varphi_{i,p} \cdot z \rceil$ of them are assigned to processor p.*

On each processor, jobs of migrating tasks are statically prioritized over jobs of fixed tasks, and jobs of same-type tasks (migrating or fixed) are prioritized against each other by EDF. With such assignment and execution phases, EDF-fm is able to guarantee bounded tardiness for each task, as long as the sum of the utilizations of any two migrating tasks that share a common processor does not exceed 1.0. Clearly, with this

utilization restriction, EDF-fm is not SRT-optimal. Nonetheless, any SRT-feasible system with a maximum per-task utilization of at most 0.5 is guaranteed to be SRT-schedulable under EDF-fm.

2.3.2 EDF-os

EDF-os (Anderson et al., 2016) is an EDF-based semi-partitioned SRT-optimal scheduling algorithm for identical multiprocessors. EDF-os also has an assignment phase and an execution phase.

In the assignment phase, EDF-os considers tasks in non-increasing-utilization order in the following two steps.

- First, it uses a worst-fit bin-packing heuristic to assign as many tasks as possible to be fixed.
- Second, it considers the remaining tasks to be assigned to processors in turn, and allocates these tasks on either one (in which case, the task is fixed) or more (in which case, the task is migrating) processors.

The following example illustrates the assignment phase of EDF-os and will be re-visited in Chapter 4.

Example 2.2. Consider scheduling the task set $\tau = \{(5,6),\ (6,9),\ (4,6),\ (2,3),\ (2,3),\ (10,30),\ (1,6)\}$ (tasks are listed in non-increasing-utilization order) on four identical processors. Figure 2.5 depicts the task assignment used by EDF-os. In the first step of the EDF-os assignment phase, the first four tasks are assigned to the four processors as fixed tasks. In the second step, the fifth task needs capacity from processors 1, 2, and 3 to be allocated, so it is a migrating task that assigns jobs to processors 1, 2, and 3. Similarly, τ_6 is a migrating task, because it has non-zero shares on both processors 3 and 4. However, the last task τ_7 is a fixed task since processor 4 is the only processor on which τ_7 has a non-zero share. For the two migrating tasks, processor 1 is the first processor of τ_5, while processor 3 is the first processor of τ_6. $\Diamond$

In the execution phase, EDF-os applies the same job-dispatching mechanism as EDF-fm, given that the fraction of jobs of each task to execute on each processor can be derived from the allocated shares in the assignment phase. We use $\varphi_{i,p}$ to denote such fraction of task τ_i on processor p. Anderson et al. (2016) showed that the following property follows from Property 2.1. This property will be referred to and reiterated in Chapter 4.

Property 2.2. *For any k consecutive jobs of a migrating task τ_i, at most $\varphi_{i,p} \cdot k + 2$ of them are assigned to processor s_p.*

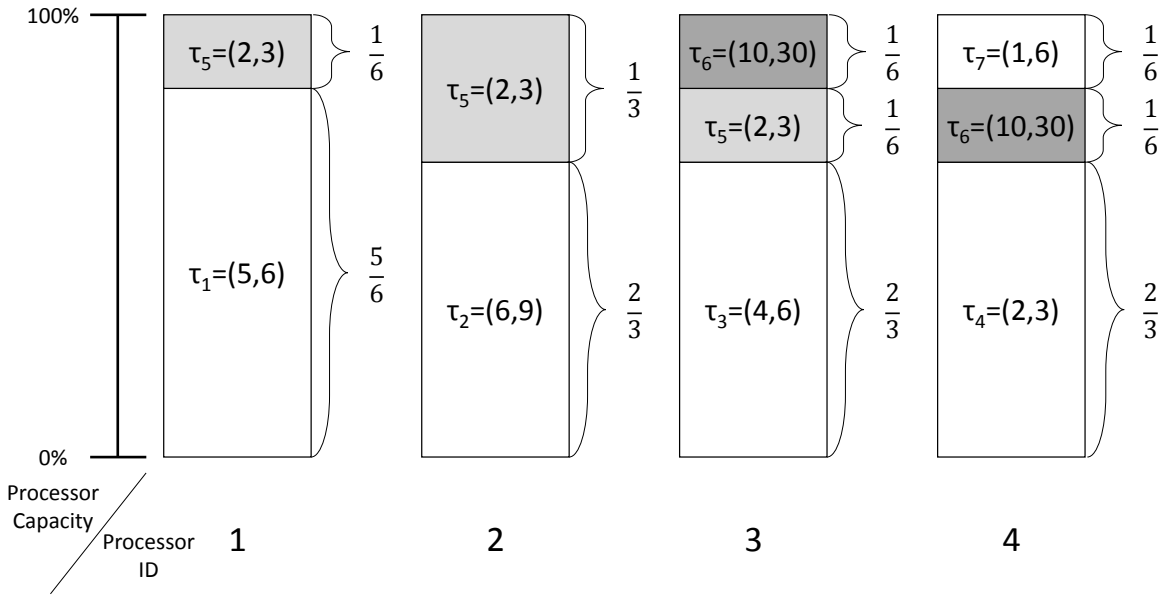


Figure 2.5: EDF-os task assignment for Example 2.2.

On each processor, the priority rules are as follows. Note that, the processor with the lowest index where a migrating task is allocated is called its *first* processor.

- Jobs of migrating tasks are statically prioritized over those of fixed tasks.
- Jobs of fixed tasks are prioritized against each other on an EDF basis.
- On a migrating task's first processor, its priority is lower than other migrating tasks, but still higher than fixed ones.

With such assignment and execution phases, EDF-os is SRT-optimal, *i.e.*, tardiness of each task is bounded under EDF-os for any SRT-feasible system.

2.3.3 EDF-ms

EDF-ms was proposed by Leontyev and Anderson (2007a) to support a multiprocessor platform consisting of processors of different speeds. Technically speaking, EDF-ms is not a semi-partitioned scheduling algorithm, because any task can migrate among at least some processors under EDF-ms, *i.e.*, no task is actually fixed to a single processor. However, EDF-ms *groups* the processors of the same speed together and performs "semi-partitioned" scheduling at the group level. It has an assignment phase similar to EDF-fm

but at the group level. As a result, in each group, there are at most two *intergroup* tasks, and those tasks executing within a single group are called *fixed* tasks (this is a different definition from that under EDF-fm or EDF-os, in which fixed tasks are tasks executing on a single processor only). In the execution phase, jobs are dispatched to groups in the same way as jobs are dispatched to processors under EDF-fm. Within each group, in which every processor is identical, jobs are prioritized by G-EDF with the exception that any job of a intergroup task will be immediately promoted to execute until completion once its *slack* reaches zero. At time t, the slack of job $\tau_{i,j}$ is defined by $(d_{i,j} - t) - (C_i - e_{i,j}(t))$, where $e_{i,j}(t)$ denotes the amount of completed work of $\tau_{i,j}$ at time t.

Leontyev and Anderson (2007a) showed that, under EDF-ms, all intergroup tasks are guaranteed to meet their implicit deadlines and the tardiness of any fix task is bounded. However, EDF-ms assumes certain utilization restrictions to facilitate the assignment phase and therefore is not SRT-optimal. Furthermore, EDF-ms requires each group to have at least two processors, *i.e.*, EDF-ms does not support platforms on which there is a processor that has a distinct speed. Nonetheless, EDF-ms is still the only work prior to this dissertation that is related to both semi-partitioned scheduling and uniform multiprocessors, so we will regard EDF-ms as a baseline in Chapter 4.

Contributions of this dissertation. All of the scheduling algorithms discussed in this section except EDF-ms apply to identical multiprocessors only, *i.e.*, they do not support a multiprocessor platform consisting of processors of different speeds. In Chapter 4, we will present two semi-partitioned scheduling algorithms designed for uniform multiprocessors where multiple speeds may exist. Compared to EDF-ms, which is able to support only certain uniform multiprocessors, we will show that both of our algorithms dominate EDF-ms in terms of schedulability.

2.4 Intra-Task Parallelism

In the sporadic task model, each task is assumed to be sequential as it usually models a piece of sequential code. However, if multiple invocations (*i.e.*, jobs) of such a piece of code are active at the same time, intra-task parallelism could be possible, even if this piece of code itself is sequential. To clarify, in this dissertation we assume tasks under the conventional sporadic task model are strictly sequential, *i.e.*, consecutive jobs of the same task cannot execute in parallel; in contrast, we will introduce the npc-sporadic task model in Chapter 5, under which jobs of the same task may execute in parallel as long as each individual job is still sequential. It

is clear that both models are the same for HRT tasks with relative deadlines at most their periods, since every job must be finished by the release time of the next one in this case. Thus, prior work exploiting intra-task parallelism under the npc-sporadic task model targeted SRT systems or HRT systems with arbitrary deadlines (the relative deadline of a task can be greater than its period).

In work on HRT systems, Baker and Baruah (2009) derived an HRT-schedulability test for arbitrary-deadline task systems on an identical multiprocessor, and both the conventional sporadic task model and the npc-sporadic task model were considered in this work. Subsequently, the impact of such intra-task parallelism was also considered by Baruah et al. (2012), Bonifaci et al. (2013), and Parri et al. (2015) in the context of HRT DAG-based task systems on an identical multiprocessor.

In work on SRT systems, Erickson and Anderson (2011) derived a response-time bound for npc-sporadic tasks systems under preemptive G-EDF on an identical multiprocessor. This is the only existing work prior to this dissertation pertaining to SRT npc-sporadic task systems.

Contributions of this dissertation. In Chapter 5, we will extend the work by Erickson and Anderson (2011) to support processors of different speeds by deriving response-time bounds for npc-sporadic task systems on a uniform multiprocessor. Furthermore, while Erickson and Anderson (2011) considered preemptive G-EDF only, we will derive such bounds under both preemptive and non-preemptive G-EDF.

2.5 DAG-based Tasks

The literature on real-time systems includes much work pertaining to the scheduling of DAG-based task systems, but most of this work assumes HRT systems and identical multiprocessor platforms. Baruah et al. (2012) provided intractability results, speed-up bounds, and EDF-schedulability tests for scheduling one sporadic DAG. Bonifaci et al. (2013) then studied the feasibility problem for multiple DAGs. Saifullah et al. (2013) conducted schedulability analysis under the *synchronous parallel task model*, which is a special case of the DAG task model. Subsequently, Li et al. (2013), Baruah (2014), Parri et al. (2015), and Jiang et al. (2016) focused on the global scheduling of DAGs, whereas Li et al. (2014), Baruah (2015a,b), Jiang et al. (2017), and Li et al. (2017) focused on the federated scheduling of DAGs. Federated scheduling for DAGs is very similar to partitioned scheduling for sporadic tasks. The difference is that the utilization of a single DAG may exceed the capacity of a single processor, and federated scheduling assigns multiple processors dedicated to solely such a DAG.

Liu and Anderson (2010) leveraged a task-transformation approach to schedule DAG-based task system using G-EDF. In contrast to most of the work discussed above, which enforces HRT-schedulability and therefore has to restrict system utilizations, Liu and Anderson (2010) derived tardiness bounds for DAG-based task systems with no utilization loss. Nonetheless, this work also assumes an identical multiprocessor platform.

Beyond the real-time systems community, DAG-based systems implemented on heterogeneous platforms have been considered before (*e.g.*, (Grandpierre et al., 1999; Bajaj and Agrawal, 2004; Stavrinides and Karatza, 2011)). However, such work focuses on one-shot, aperiodic DAG-based jobs, rather than periodic or sporadic DAG-based task systems. Moreover, real-time issues are considered only obliquely from the perspectives of job admission control or job makespan minimization.

Contributions of this dissertation. In Chapter 6, we will extend the task-transformation techniques by Liu and Anderson (2010) to heterogeneous platforms where processors may have different functionalities. Furthermore, in contrast to more focusing on per-node tardiness bounds as in (Liu and Anderson, 2010), we will more focus on per-DAG end-to-end response-time bounds. In addition, we will further study deadline-setting techniques in this context and present a LP-based method to set deadlines in order to improve the end-to-end response-time bounds of DAGs. The preliminary version of this work published in (Yang et al., 2016) also led to follow-up work by Dong et al. (2017), which presented an alternative technique for deriving tighter end-to-end response-time bounds in certain scenarios.

2.6 Compositional Real-Time Systems

Existing work in compositional real-time systems has been directed at both uniprocessors and multiprocessors.

In work on uniprocessors, Mercer et al. (1994) proposed a mechanism that abstracts the notion of a processor-capacity reservation as a uniprocessor with reduced speed. Abeni and Buttazzo (1998) proposed the *constant bandwidth server (CBS)* to integrate HRT tasks and multimedia applications with soft timing requirements in a single system. Lipari and Baruah (2001) then extended CBS to a hierarchical scheduling framework. Mok et al. (2001) proposed the *bounded-delay partition* model, in which a partition specified by (α, Δ) provides processor supply between $\alpha \cdot t$ and $\alpha \cdot (t - \Delta)$ from time 0 to time t. Based on the bounded-delay partition model, Lipari and Bini (2003) derived the "best" setting of parameters for a given

application. Shin and Lee (2003) proposed the *periodic resource (PR)* model model, in which a virtual processor (VP) is specified by the parameters (Π, Θ), with the interpretation that Θ time units of processor time is guaranteed to the supported component every Π time units. Easwaran et al. (2007) then extended the PR model to the *explicit deadline periodic (EDP)* resource model by adding a parameter Δ to the PR model, with the interpretation that the supply must be provided in the first Δ time units in each period.

In work on multiprocessors, Leontyev and Anderson (2009) initially proposed MP form to schedule each component using at most one partially available processor in SRT systems. Shin et al. (2008) proposed the *multiprocessor periodic resource (MPR)* model, in which the supply allocated to a component is specified by (Π, Θ, m'), with the interpretation that Θ time units of processor time is guaranteed to the component every Π time units with at most m' VPs providing allocation in parallel. Easwaran et al. (2009) derived a cluster-based hierarchical scheduler by applying the MPR model. Burmyakov et al. (2014) extended the MPR model by providing information of resource allocation at each degree of parallelism. Xu et al. (2015) extended the MPR model to the *deterministic MPR (DMPR)* model by requiring VPs to be allocated in MP form, and proposed a cache-aware analysis framework.

In much of the discussed work, a *supply bound function (SBF)* is provided to characterize the minimum resource allocation of a component, in order to perform schedulability analysis. Furthermore, Bini et al. (2009b) proposed the *multi supply function (MSF)* that provides a separate SBF for each VP on a virtual multiprocessor platform. Subsequently, Bini et al. (2009a) proposed the *parallel supply function (PSF)*, which is strictly more powerful than MSF. We will review PSF in detail in Chapter 7, in which PSF is heavily used. To the best of our knowledge, PSF is the most expressive means of characterizing resource-allocation supply on multiprocessors.

Contributions of this dissertation. In Chapter 7, we will establish the dominance of MP form in terms of the fundamental supply-bound functions characterized by PSF for virtual multiprocessor platforms. Compared to the work by Leontyev and Anderson (2009) pertaining to SRT systems, our work applies to HRT systems as well. Compared to the work by Xu et al. (2015), which assumes a common, synchronous period among all VPs, our work applies to virtual platforms consisting of VPs that may have different, asynchronous periods.

2.7 Chapter Summary

In this chapter, we reviewed prior work on several topics related to this dissertation, namely feasibility and EDF scheduling for HRT systems on uniform multiprocessors, tardiness bounds for SRT systems, semi-partitioned scheduling, intra-task parallelism, DAG-based tasks, and compositional real-time systems. In light of such prior work, we emphasized the difference between the work in this dissertation and prior work and briefly highlighted the contributions of this dissertation in the context of the existing literature with respect to each topic.

CHAPTER 3: GLOBAL EDF SCHEDULING ON UNIFORM PLATFORMS[1]

In this chapter, we study the problem of whether G-EDF is SRT-optimal on uniform multiprocessors. Devi and Anderson (2008) have shown that G-EDF is SRT-optimal on identical multiprocessors, no matter whether preemptive or non-preemptive scheduling is assumed. In light of the fact that any identical multiprocessor is a special case of uniform multiprocessors, one conjecture following the work by Devi and Anderson (2008) is that G-EDF is also SRT-optimal on uniform multiprocessors. However, such an extension was found to be surprisingly difficult and this conjecture remained open until our work, summarized in this chapter, closed it.

The key difficulty faced when trying to extend prior tardiness analysis for identical multiprocessors to uniform ones is that, in the uniform case, tasks can execute on processors that are "too slow." The specific problematic property required in the prior analysis is the following.

> **(P)** If any job $\tau_{i,j}$ executes continuously, then it must complete within T_i time units, regardless of the processor on which it executes.

Clearly, (P) can be violated on a uniform multiprocessor, if $\tau_{i,j}$ executes entirely on processors of speed less than u_i.

It is tempting to obviate all problematic issues pertaining to Property (P) by simply enforcing scheduling policies that uphold it. Such an approach was taken by Tong and Liu (2016), who considered a variant of G-EDF in which each task τ_i is only allowed to execute on processors with speed at least u_i. However, such a requirement results in non-optimal scheduling. For example, Tong and Liu's G-EDF variant is not able to correctly schedule a set of two tasks on two processors such that $u_1 = 2$, $u_2 = 2$, $s_1 = 3$, and $s_2 = 1$. From (3.5) and (3.6), which together are a SRT-feasibility condition, as will be shown later in Section 3.2, we see

[1]Contents of this chapter previously appeared in preliminary form in the following papers:

Yang, K. and Anderson, J. (2015a). On the soft real-time optimality of global EDF on multiprocessors: From identical to uniform heterogeneous. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Yang, K. and Anderson, J. (2016b). Tardiness bounds for global EDF scheduling on a uniform multiprocessor. In *Proceedings of the 7th International Real-Time Scheduling Open Problems Seminar*, pages 3–4.

Yang, K. and Anderson, J. (2017). On the soft real-time optimality of global EDF on uniform multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 319–330.

that this task set is SRT-feasible. However, under their algorithm, a task τ_i can only execute on a processor of speed at least u_i, so both tasks in this example must exclusively execute on the processor with speed $s_1 = 3$. That processor will be over-utilized if each task releases jobs as soon as possible and always executes for its worst-case cost, since $u_1 + u_2 = 4 > s_1 = 3$.

In other work, we successfully eliminated the need for Property (P) by relaxing the task model to allow consecutive jobs of the same task to execute in parallel. Under this relaxed task model, we were able to establish the SRT optimality of G-EDF on uniform platforms (Yang and Anderson, 2014a). Details of this work will be summarized in Chapter 5.

For the sequential task model being considered in this chapter, we also found that Property (P) is not necessary if the underlying uniform platform has only two processors (Yang and Anderson, 2015a). However, we believe that it is unlikely that the particular proof strategy used in (Yang and Anderson, 2015a) can be extended to the more general m-processor case.

To clearly place our contribution in its proper context in light of this prior work, we emphasize here several assumptions made hereafter in this chapter:

- we are interested in *any* SRT-feasible task set, *i.e.*, no constraints on task utilizations other than (3.5) and (3.6) are assumed;
- intra-task parallelism is strictly forbidden, *i.e.*, jobs of the same task must execute in sequence;
- the uniform platform may have m processors, where m can be any positive integer value.

We will show that, in spite of being SRT-optimal for identical multiprocessors (Devi and Anderson, 2008), non-preemptive G-EDF is not SRT-optimal for uniform multiprocessors, by providing a counterexample where a SRT-feasible system may experience unbounded tardiness under non-preemptive G-EDF scheduling. In contrast, preemptive G-EDF is indeed SRT-optimal for uniform multiprocessors. We will prove this by deriving a tardiness bound for an arbitrary SRT-feasible system under preemptive G-EDF scheduling; the proof strategy is significantly different from that in (Devi and Anderson, 2008) for identical multiprocessors.

Organization. In the following sections, we provide needed background and notation (Section 3.1), establish a necessary and sufficient SRT-feasibility condition (Section 3.2), formally define the two considered variants of G-EDF (Section 3.3), and then disprove the SRT-optimality for non-preemptive G-EDF (Section 3.4) and prove the SRT-optimality for preemptive G-EDF (Section 3.5).

3.1 System Model

In this chapter, our focus is the uniform multiprocessor model. Specifically, we consider the scheduling of a set τ of n sequential tasks on a uniform platform π consisting of m processors, where the processors are indexed by their speeds in non-increasing order, *i.e.*, $s_i \geq s_{i+1}$ for $i = 1, 2, \ldots, m-1$. We denote the sum of k largest speeds on π as $S_i = \sum_{i=1}^{k} s_i$ for $k = 1, 2, \ldots, m$. Furthermore, we assume $m \geq 2$, for otherwise, uniprocessor analysis can be applied. We also assume $n \geq m$, for otherwise, there is no point in ever scheduling any task on any of the $m - n$ slower processors, so m and n can conceptually be deemed as equal in this case.

We consider the conventional *sporadic* tasks. A sporadic task τ_i releases a sequence of *jobs* with a minimum separation of T_i time units between invocations. The parameter T_i is called the *period* of τ_i. τ_i also has a *worst-case execution requirement* C_i, which is defined as the maximum execution time of any one job (invocation) of τ_i on a unit-speed processor. We let $C_{\max} = \max\{C_i \mid 1 \leq i \leq n\}$. The *utilization* of task τ_i is given by $u_i = C_i/T_i$. We assume that tasks are indexed in non-increasing order by utilization, *i.e.*, $u_i \geq u_{i+1}$ for $i = 1, 2, \ldots, n-1$. We denote the sum of k largest utilizations in τ as $U_i = \sum_{i=1}^{k} u_i$ for $k = 1, 2, \ldots, n$. Furthermore, we denote the ratio between the largest and the smallest utilizations as $\rho = u_1/u_n$. As for scheduling, we assume that deadlines are *implicit*, *i.e.*, each task τ_i has a *relative deadline* parameter equal to its period T_i. Furthermore, under the conventional sporadic task model, tasks are *sequential* and intra-task parallelism is not allowed. That is, an invocation of a task cannot commence execution until all previous invocations of that task complete. In this chapter, we assume that time is continuous.

The j^{th} job (or invocation) of task τ_i is denoted as $\tau_{i,j}$. Job $\tau_{i,j}$ has a release time denoted $r_{i,j}$, an absolute deadline denoted $d_{i,j} = r_{i,j} + T_i$, and a completion (or finish) time denoted $f_{i,j}$. The *tardiness* of job $\tau_{i,j}$ is defined by $\max\{0, f_{i,j} - d_{i,j}\}$ and its *response time* by $f_{i,j} - r_{i,j}$. The *tardiness* of *task* τ_i in some schedule is the maximum tardiness of any of its jobs in that schedule. A job is *pending* if it is released but has not completed, and is *ready* if it is pending and all preceding jobs of the same task have completed.

Ideal schedule. We define an *ideal multiprocessor* $\pi_{\mathcal{I}}$ for the task set τ as one that consists of n uniform processors where the speeds of the n processors exactly match the utilizations of the n tasks in τ, respectively, *i.e.*, the speed of the i^{th} processor is $s_i^{\mathcal{I}} = u_i$ for $i = 1, 2, \ldots, n$. We define the *ideal schedule* $\mathcal{I}$ to be the partitioned schedule for τ on $\pi_{\mathcal{I}}$, where each task τ_i in τ is assigned to the processor of speed $s_i^{\mathcal{I}}$. Then, in $\mathcal{I}$, every job in τ commences execution at its release time and completes execution within one period (it exactly

executes for one period if and only if its actual execution requirement matches its worst-case execution requirement). Thus, all deadlines are met in $\mathcal{I}$.

Definition of lag**.** Let $\mathsf{A}(\mathcal{S},\tau_i,t_1,t_2)$ denote the cumulative processor capacity allocated to task τ_i in an arbitrary schedule $\mathcal{S}$ within the time interval $[t_1,t_2)$. By the definition of the ideal schedule $\mathcal{I}$,

$$\mathsf{A}(\mathcal{I},\tau_i,t_1,t_2) \leq u_i \cdot (t_2 - t_1). \tag{3.1}$$

Also, if τ_i releases jobs periodically and every job's actual execution requirement equals its worst case of C_i, then for any t_1 and t_2 such that $r_{i,1} \leq t_1 \leq t_2$,

$$\mathsf{A}(\mathcal{I},\tau_i,t_1,t_2) = u_i \cdot (t_2 - t_1). \tag{3.2}$$

For an arbitrary schedule $\mathcal{S}$, we denote the difference between the allocation to a task τ_i in $\mathcal{I}$ and in $\mathcal{S}$ within time interval $[0,t)$ as

$$\mathsf{lag}(\tau_i,t,\mathcal{S}) = \mathsf{A}(\mathcal{I},\tau_i,0,t) - \mathsf{A}(\mathcal{S},\tau_i,0,t). \tag{3.3}$$

The lag function captures the allocation difference between an arbitrary actual schedule $\mathcal{S}$ and the ideal schedule $\mathcal{I}$. If $\mathsf{lag}(\tau_i,t,\mathcal{S})$ is positive, then $\mathcal{S}$ has performed less work on τ_i until time t, *i.e.*, τ_i is "under-allocated," while if $\mathsf{lag}(\tau_i,t,\mathcal{S})$ is negative, then τ_i is "over-allocated." Also, for any two time instants t_1 and t_2 such that $t_1 \leq t_2$, we have

$$\mathsf{lag}(\tau_i,t_2,\mathcal{S}) = \mathsf{lag}(\tau_i,t_1,\mathcal{S}) + \mathsf{A}(\mathcal{I},\tau_i,t_1,t_2) - \mathsf{A}(\mathcal{S},\tau_i,t_1,t_2). \tag{3.4}$$

At a given time instant t, we say that a *task* is *pending* if it has any pending jobs at time t. If task τ_i is pending at time t, then it has exactly one ready job $\tau_{i,j}$ at time t. The deadline of that job is called the *effective* deadline of τ_i at time t and is denoted $d_i(t) = d_{i,j}$. Similarly, the *effective* release time of τ_i at time t is denoted $r_i(t) = r_{i,j}$. Because deadlines are implicit, $d_i(t) = r_i(t) + T_i$. Also, $r_i(t) \leq t$ holds, for otherwise, $\tau_{i,j}$ would not be ready at time t. The following lemma gives a sufficient lag-based condition for a task to be pending.

Lemma 3.1. *If* $\mathsf{lag}(\tau_i,t,\mathcal{S}) > 0$*, then* τ_i *is pending at time* t *in* $\mathcal{S}$*.*

Proof. Suppose that $\mathsf{lag}(\tau_i,t,\mathcal{S}) > 0$ holds but τ_i is not a pending task at time t in $\mathcal{S}$. Then, all jobs of τ_i released at or before t have completed by time t. Thus, letting W denote the total actual execution requirement of all such jobs, we have $\mathsf{A}(\mathcal{S},\tau_i,0,t) = W$. In the ideal schedule $\mathcal{I}$, only released jobs can be scheduled and will not execute for more than their actual execution requirement. Thus, $\mathsf{A}(\mathcal{I},\tau_i,0,t) \leq W$ holds as well. By (3.3), these facts imply $\mathsf{lag}(\tau_i,t,\mathcal{S}) = \mathsf{A}(\mathcal{I},\tau_i,0,t) - \mathsf{A}(\mathcal{S},\tau_i,0,t) \leq 0$. This contradicts our assumption that $\mathsf{lag}(\tau_i,t,\mathcal{S}) > 0$ holds. □

3.2 A Necessary and Sufficient SRT-Feasibility Condition.

For HRT task sets, Funk et al. (2001) showed that a set of implicit-deadline periodic tasks is feasible on a uniform platform if and only if the following constraints hold:

$$U_n \leq S_m, \tag{3.5}$$

$$U_k \leq S_k, \text{ for } k = 1,2,\ldots,m-1. \tag{3.6}$$

It can be shown that this constraint set is also a feasibility condition for implicit-deadline *sporadic* tasks. Furthermore, the sufficiency of this constraint set for HRT task sets implies its sufficiency for SRT task sets. In fact, these constraints are necessary for SRT task sets as well. To see this, note that if $U_n > S_m$ holds (contrary to (3.5)), then the total workload over-utilizes the platform, so some task will be increasingly tardy without bound if tasks release jobs as soon as possible and always execute for their worst-case costs. Furthermore, if $U_k > S_k$ holds (contrary to (3.6)), then the set of k highest-utilization tasks will be "under-allocated" at every time instant if they release jobs as soon as possible and always execute for their worst-case costs. This is because k tasks can be allocated to at most k processors at any time instant and the sum of the speeds of *any* k processors can be at most S_k. Thus, $\sum_{i=1}^{k} \mathsf{lag}(\tau_i,t,\mathcal{S})$ will increase without bound, which implies that $\mathsf{lag}(\tau_i,t,\mathcal{S})$ will increase without bound for some i. This implies that task τ_i will be increasingly tardy without bound.

To summarize, (3.5) and (3.6) are also a necessary and sufficient feasibility condition for SRT task sets. Therefore, when henceforth referring to this constraint set as a feasibility condition, we do not need to further specify whether this is meant for HRT or SRT task sets.

3.3 Preemptive and Non-Preemptive G-EDF Scheduling on Uniform Multiprocessors.

From a scheduling point of view, uniform platforms differ from identical ones in a significant way: on a uniform platform, besides *which* tasks are scheduled at any time, the scheduler must also decide *where* they are scheduled, because different processors may have different speeds. Thus, we must refine the notion of G-EDF scheduling to be clear about where tasks are scheduled. In particular, we consider the following two G-EDF scheduling algorithms.

> **Preemptive G-EDF:** If at most m jobs are ready, then all ready jobs are scheduled; otherwise, the m ready jobs with earliest deadlines are scheduled. At any time, the ready job with the k^{th} earliest deadline is scheduled on the k^{th} fastest processor for any k. (Note that this implies that a job may migrate from one processor to another during its execution.) Deadline ties are broken arbitrarily.
>
> **Non-Preemptive G-EDF:** Any job enters a deadline-prioritized queue once it becomes ready. Whenever this queue is non-empty and some processor(s) are idle, the ready job at the head of the queue is dequeued and scheduled on the fastest idle processor for execution without *preemption* nor *migration* until its completion.

Note that, when a uniform multiprocessor reduces to an identical one (*i.e.*, $s_i = 1.0$ for all i), the preemptive and non-preemptive G-EDF schedulers above also reduce to simpler and more intuitive preemptive and non-preemptive G-EDF schedulers for identical multiprocessors.

3.4 Tardiness Increasing without Bound under Non-Preemptive G-EDF

Devi and Anderson (2008) proved that both preemptive and non-preemptive G-EDF are SRT-optimal on identical multiprocessors. If a simple extension to uniform multiprocessors existed for the analysis in (Devi and Anderson, 2008), they would have been SRT-optimal on uniform multiprocessors too. Unfortunately, this is not true.

In this section, we prove that non-preemptive G-EDF is not SRT-optimal on uniform platforms by proving that no *work-conserving* non-preemptive scheduler is SRT-optimal on uniform platforms by giving a counterexample. We begin with formally defining "non-preemptive" and "work-conserving" in this context.

Non-preemptive. A non-preemptive scheduler schedules ready jobs, and once a job is scheduled, it continually executes without preemption until it completes. In this section, we further define that under non-preemptive scheduling, once a job is scheduled, it continually executes *on the processor on which it was scheduled* without preemption until it completes, *i.e.*, non-preemptivity means no preemption and no *migration* occurs within a single job. This implicitly holds for any non-preemptive scheduler on identical multiprocessors, where every processor is the same and therefore there is no point to migrating a job that is currently executing; however, we do have to clarify this here, since the processors in a uniform platform could be of different speeds.

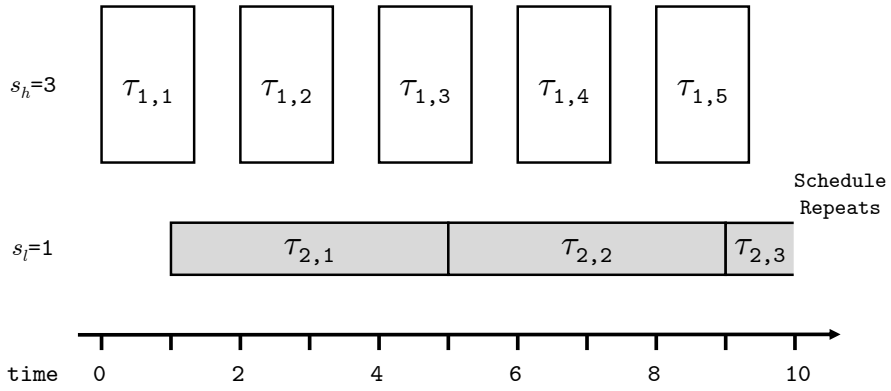
Work-conserving. A *work-conserving* scheduler prevents the situation where there is at least one processor that is idle, and at least one task that has a incomplete ready job but is not scheduled, *i.e.*, whenever a task could be scheduled *somewhere*, it is scheduled. Both preemptive and non-preemptive G-EDF are clearly a work-conserving.

The counterexample. We consider a two-processor uniform platform $\pi = \{s_h = 3, s_l = 1\}$ and a task set of two tasks, $\tau_1 = (4,2)$ and $\tau_2 = (4,2)$, to be scheduled on π. Also, we consider the situation where τ_1 releases its first job at time 0 and then releases jobs as soon as possible, and τ_2 releases its first job at time 1 and then releases jobs as soon as possible. Furthermore, we assume every job has an execution requirement that matches its worst-case execution requirement.

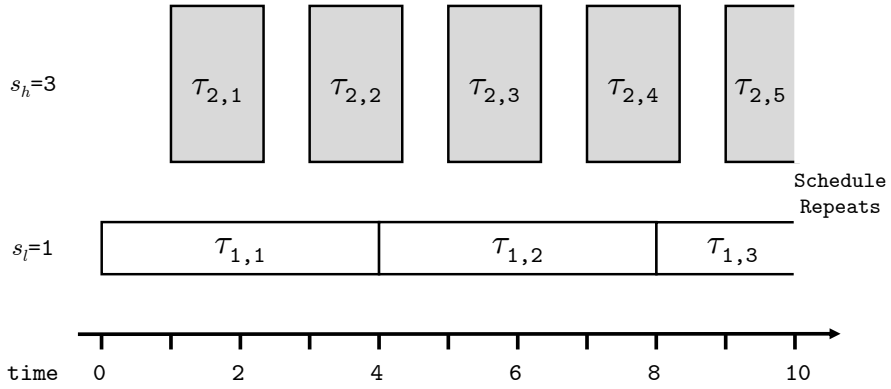
At time 0, $\tau_{1,1}$ is released, a work-conserving scheduler must schedule it on either s_h or s_l.

Case 1: $\tau_{1,1}$ is scheduled on s_h (Figure 3.1 (a)). Then, $\tau_{1,1}$ continuously executes on s_h until time 1.33. Therefore, at time 1 when $\tau_{2,1}$ is released, s_l and only s_l is available. Thus, a work-conserving scheduler must schedule $\tau_{2,1}$ on s_l, where $\tau_{2,1}$ continuously executes until time 5, which means both $\tau_{1,2}$ (released at time 2) and $\tau_{1,3}$ (released at time 4) must be scheduled on s_h and each of them continuously executes on s_h for 1.33 time units. Thus, at time 5 when $\tau_{2,1}$ completes and $\tau_{2,2}$ is ready, s_l and only s_l is available, which means that a work-conserving scheduler must scheduler $\tau_{2,2}$ on s_l where $\tau_{2,2}$ continuously executes until time 9. This pattern repeats in the schedule. Figure 3.1 (a) shows the schedule. Observe that τ_2 is always scheduled on s_l and therefore becomes unboundedly tardy.

Case 2: $\tau_{1,1}$ is scheduled on s_l (Figure 3.1 (b)). Then, $\tau_{1,1}$ continuously executes on s_h until time 4, which means both $\tau_{2,1}$ (released at time 1) and $\tau_{2,2}$ (released at time 3) must be scheduled on s_h and each of them continuously executes on s_h for 1.33 time units. Thus, at time 4 when $\tau_{1,1}$ completes and $\tau_{1,2}$ is ready, s_l



(a) Case 1.



(b) Case 2.

Figure 3.1: Counterexample schedules.

and only s_l is available, which means that a work-conserving scheduler must scheduler $\tau_{1,2}$ on s_l where $\tau_{1,2}$ continuously executes until time 8. This pattern repeats in the schedule. Figure 3.1 (b) shows the schedule. Observe that τ_1 is always scheduled on s_l and therefore becomes unboundedly tardy.

Thus, in this system, under any work-conserving non-preemptive scheduler, there must be one task that has unbounded tardiness. However, by (4.29) and (4.30), this system is actually feasible. Figure 3.2 shows a feasible schedule for this system where all deadlines are met.

In this counterexample, each task releases subsequent jobs as soon as possible, so it is valid not only for sporadic tasks, but also for periodic tasks where the two tasks have phases 0 and 1, respectively. Also,

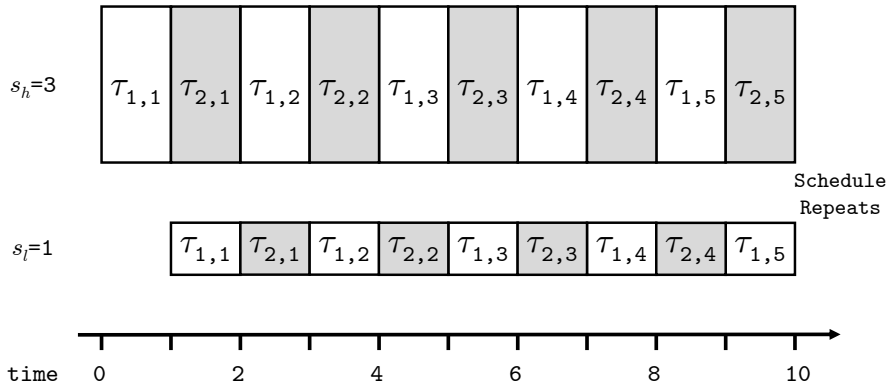


Figure 3.2: Feasible schedule.

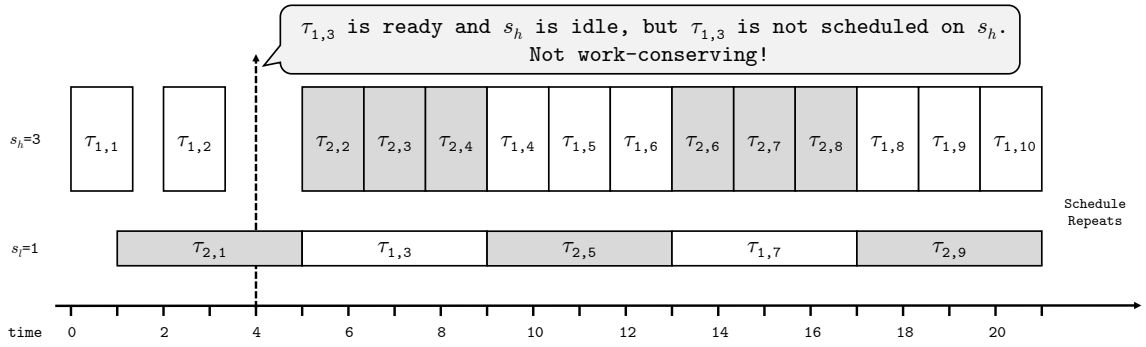


Figure 3.3: A non-preemptive schedule for the system in Section 3.4. Note that the deadline tardiness is upper bounded by 3 time units.

the two-processor uniform platform considered in this section is a special case for the more general uniform platform where the number of processors is arbitrary. Thus, the following theorem holds.

Theorem 3.1. *No work-conserving non-preemptive scheduler is SRT-optimal for sequential sporadic or periodic tasks on uniform multiprocessors.*

One might wonder whether if this system with this job release pattern is SRT-feasible under the non-preemptive restriction, *i.e.*, under non-preemptive scheduling, whether it is possible to have bounded tardiness for every task in this system. In fact, this system with this job release pattern is indeed SRT-feasible for non-preemptive scheduling. For example, Figure 3.3 is a non-preemptive schedule for this system, and deadline tardiness is at most 3 time units.

The key in the schedule in Figure 3.3 is that it is not work-conserving. At time 4, $\tau_{1,3}$ is ready and s_h is idle, but in this schedule $\tau_{1,3}$ is not scheduled until time 5. At time 5, when $\tau_{2,1}$ has completed on s_l, we schedule $\tau_{2,2}$ on s_h, and schedule $\tau_{1,3}$ on s_l. Then, the schedule can be repeated in a way that the two tasks are scheduled on the faster processor in turn. As shown in Figure 3.3, the maximum tardiness of τ_1 is 3 time units ($\tau_{1,3}$, $\tau_{1,7}$, ...); the maximum tardiness of τ_2 is 2 time units ($\tau_{2,1}$, $\tau_{2,5}$, $\tau_{2,9}$,...).

3.5 Tardiness Bounds under Preemptive G-EDF

With the above negative result regarding SRT-optimality of non-preemptive G-EDF, we turn our attention to preemptive G-EDF and consider the question: is preemptive G-EDF SRT-optimal on uniform multiprocessors? In this section, we answer this question by proving a tardiness bound for any feasible sporadic task system on a uniform multiprocessor. Furthermore, we actually prove a tardiness bound for a more general task model—VPP tasks—as introduced next. Also, we omit "preemptive" and assume all references to G-EDF in Section 3.5 to mean the preemptive G-EDF scheduling algorithm as defined in Section 3.3.

3.5.1 Varying-Period Periodic Tasks

An implicit-deadline *varying-period periodic (VPP)* task τ_i^V has a pre-defined utilization u_i^V and also releases a sequence of jobs. However, in contrast to the ordinary periodic task model, each VPP job $\tau_{i,j}^V$ has its own worst-case execution requirement, denoted $C_{i,j}$. After its first invocation, a VPP task τ_i^V will release each job $\tau_{i,j+1}^V$ *exactly* $T_{i,j} = C_{i,j}/u_i$ time units after $\tau_{i,j}^V$'s release. Also, each job $\tau_{i,j}^V$ has a deadline $T_{i,j}$ time units after its release. For each VPP task τ_i^V, C_i^V is defined as $C_i^V = \max\{C_{i,j} \mid j \geq 1\}$ and T_i^V is defined as $T_i^V = \max\{T_{i,j} \mid j \geq 1\}$. Note that an ordinary periodic task τ_i is a special case of a VPP task where $C_{i,j} = C_{i,k}$ holds (and hence $T_{i,j} = T_{i,k}$ holds) for any j and k. In accordance to the specification of periodic and sporadic tasks, we also specify a VPP task by $\tau_i^V = (C_i^V, T_i^V)$ where $T_i^V = C_i^V/u_i$.

Sporadic tasks. In fact, not only an ordinary periodic task but also a *sporadic* task is a special case of a VPP task set. We show this by showing that any instance[2] of a sporadic task $\tau_i = (C_i, T_i)$ can be viewed as an instance of a VPP task $\tau_i^V = (C_i^V, T_i^V)$ where $C_i = C_i^V$ and $T_i = T_i^V$. This transformation is depicted in Figure 3.4 and explained next.

[2] An instance of a task is defined by a set of *concrete* job release times and *actual* execution requirements that satisfy the specification of the task.

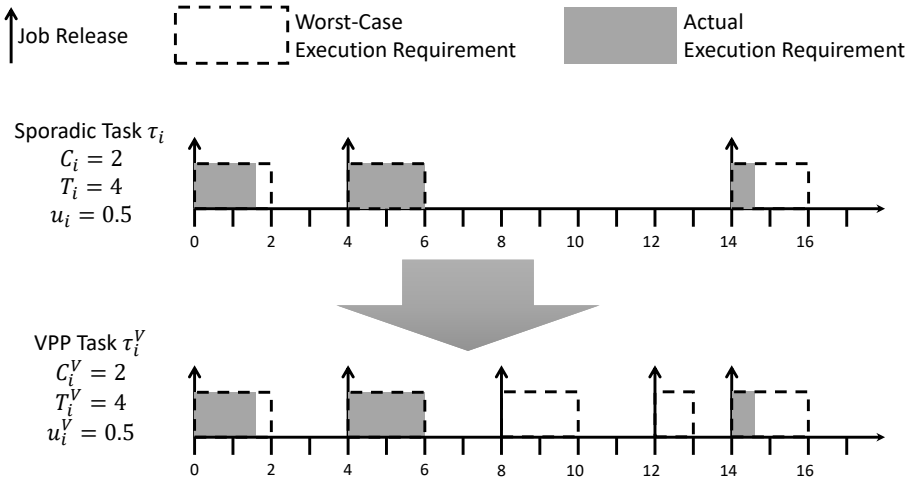


Figure 3.4: Transforming a sporadic task into a VPP task.

In the sporadic task model, a given instance of a task $\tau_i = (C_i, T_i)$ might have two consecutive jobs that have a release separation of more than T_i time units. Let $\tau_{i,j}$ and $\tau_{i,j+1}$ be such two jobs, *i.e.*, $r_{i,j+1} - r_{i,j} > T_i$. Let us denote $r_{i,j+1} - r_{i,j}$ as $(k+1)T_i + Q$ where k is an integer such that $k \geq 0$ and Q is a real number such that $0 \leq Q < T_i$ (k and Q can be easily calculated from $r_{i,j+1} - r_{i,j}$). To see this is indeed an instance of a VPP task, we add k jobs, with the ℓ^{th} one released at time $r_{i,j} + \ell \cdot T_i$ for $1 \leq \ell \leq k$, all with a worst-case execution requirement of C_i, plus an additional job, if $Q > 0$, released at time $r_{i,j+1} - Q$, where this job has a worst-case execution requirement of $Q \cdot \frac{C_i}{T_i}$. We do this whenever $r_{i,j+1} - r_{i,j} > T_i$ holds for two consecutive job releases $r_{i,j}$ and $r_{i,j+1}$. The resulting job release times fit the specification of VPP task $\tau_i^V = (C_i^V, T_i^V)$ where $C_i^V = C_i$ and $T_i^V = T_i$. Given the job release time, in order to obtain a instance of τ_i^V, we can simply define the *actual* execution requirement of each added job to be zero and the resulting instance of VPP task τ_i^V is indeed the instance of sporadic task τ_i considered at the beginning of this paragraph.

Thus, the VPP task model is a more general model than the sporadic task model and the following claim is true.

Claim 3.1. *The tardiness of any sporadic task system satisfying* (3.5) *and* (3.6) *is bounded, if the tardiness of any VPP task system satisfying* (3.5) *and* (3.6) *is bounded.*[3]

Henceforth, we consider a VPP task set in the remainder of this section and omit the superscript "V".

[3]Provided the superscript "V" is added accordingly in (3.5) and (3.6).

Furthermore, the following theorem shows that, under G-EDF, any job(s) executing less than their worst-case execution requirement will not cause any tardiness increase.

Theorem 3.2. *For a given VPP task set τ, let $\mathcal{S}$ denote a G-EDF schedule, and let $\mathcal{S}'$ denote a corresponding G-EDF schedule where some job(s) have less execution requirement ("corresponding" means that $\mathcal{S}$ and $\mathcal{S}'$ include exactly the same jobs, released at exactly the same time instants). Then, no job finishes later in $\mathcal{S}'$.*

Proof. We prove the theorem by considering jobs inductively in deadline order. (We assume that deadline ties are broken the same way in both schedules.) Note that, under G-EDF scheduling, the scheduling of a given job is not impacted by any lower priority jobs.

Base case. The highest-priority job cannot finish later in $\mathcal{S}'$. In particular, because this job has the highest priority, it will execute continuously on the fastest processor once released.

Inductive step. Let $\mathcal{J}$ denote the set of k highest-priority jobs, and assume that these jobs do not finish later in $\mathcal{S}'$. Also, let J denote the $(k+1)^{st}$ highest-priority job. We show that J also does not finish later in $\mathcal{S}'$.

Because no job in $\mathcal{J}$ finishes later in $\mathcal{S}'$, at any time instant t, the number of *ready* jobs in $\mathcal{S}'$ does not increase[4] in comparison to $\mathcal{S}$. Therefore, up to any time instant t after its release, J is allocated in $\mathcal{S}'$ no less computing capacity than in $\mathcal{S}$, unless J has completed in $\mathcal{S}'$ but not in $\mathcal{S}$ prior to time t. Finally, because J's execution requirement is no greater in $\mathcal{S}'$ than in $\mathcal{S}$, it cannot finish later in $\mathcal{S}'$. □

Theorem 3.2 implies the following claim.

Claim 3.2. *The tardiness of any instance of a VPP task system is upper-bounded by the tardiness of the instance where all jobs execute for their worst-case execution requirement of this VPP task system.*

By Claims 3.1 and 3.2, in order to derive a tardiness bound for any feasible sporadic task set, we just need to derive a tardiness bound for all VPP task set satisfying (3.5) and (3.6), and all jobs can be assumed to execute for exactly their worst-case execution requirement. Such a tardiness bound will be shown next in Section 3.5.2.

[4]To see this, note that, if a job J' in $\mathcal{J}$ is not ready in $\mathcal{S}$ but ready in $\mathcal{S}'$ at some time t, then J' must be *pending* at time t in both $\mathcal{S}$ and $\mathcal{S}'$, and some preceding job of the same task must be ready in $\mathcal{S}$ but completed in $\mathcal{S}'$.

3.5.2 Deriving Tardiness Bounds

In this section, we prove tardiness bounds for an arbitrary feasible VPP task set satisfying (3.5) and (3.6) on a uniform platform π, under the following assumption.

(A) Every job of any task executes for its worst-case execution requirement of C_i.

Our objective is to derive tardiness bounds when the G-EDF scheduler is used to schedule τ. We do so by reasoning about lag values in an arbitrary G-EDF schedule $\mathcal{S}$ for τ. The concept of lag is useful for our purposes because a task that has positive lag at one of its deadlines will have a tardy job. Focusing on VPP task sets where Assumption (A) holds facilitates much of the lag-based reasoning that is needed.

Properties of lag values and deadlines. We begin by proving a number of properties concerning lag values and deadlines and relationships between the two. The first such property is given in the following lemma.

Lemma 3.2. *If task τ_i is pending at time t in $\mathcal{S}$, then its effective deadline $d_i(t)$ has the following relationship with $\mathsf{lag}(\tau_i,t,\mathcal{S})$.*

$$t - \frac{\mathsf{lag}(\tau_i,t,\mathcal{S})}{u_i} < d_i(t) \leq t - \frac{\mathsf{lag}(\tau_i,t,\mathcal{S})}{u_i} + T_i \tag{3.7}$$

Proof. Let $e_{i,j}(t)$ denote the remaining execution requirement for the ready job $\tau_{i,j}$ of τ_i at time t in $\mathcal{S}$. Because $\tau_{i,j}$ is ready at time t, it has not finished execution by then, so

$$0 < e_{i,j}(t) \leq C_{i,j}. \tag{3.8}$$

Furthermore, all jobs of τ_i prior to $\tau_{i,j}$ have completed by time t in $\mathcal{S}$. Let W denote the total execution requirement for all of these jobs. Then, given Assumption (A),[5]

$$\mathsf{A}(\mathcal{S},\tau_i,0,t) = W + C_{i,j} - e_{i,j}(t). \tag{3.9}$$

[5]Without Assumption (A), $\tau_{i,j}$ may execute in total for less than its worst-case execution requirement of C_i, and therefore only "$\leq$" can be claimed in (3.9).

Now consider the ideal schedule $\mathcal{I}$. In it, all jobs of τ_i prior to $\tau_{i,j}$ have complete by time $r_i(t) \leq t$. Given Assumption (A), within $[r_i(t), t)$, $\mathcal{I}$ continuously[6] executes job $\tau_{i,j}$ at a rate of u_i. Thus,

$$\mathsf{A}(\mathcal{I}, \tau_i, 0, t) = W + (t - r_i(t))u_i. \tag{3.10}$$

Therefore, an expression for $\mathsf{lag}(\tau_i, t, \mathcal{S})$ can be derived as follows.

$$\begin{aligned}
\mathsf{lag}(\tau_i, t, \mathcal{S}) = & \ \{\text{by (3.3)}\} \\
& \mathsf{A}(\mathcal{I}, \tau_i, 0, t) - \mathsf{A}(\mathcal{S}, \tau_i, 0, t) \\
= & \ \{\text{by (3.9) and (3.10)}\} \\
& (t - r_i(t))u_i - (C_i - e_{i,j}(t)) \\
= & \ \{\text{because } d_i(t) = r_i(t) + T_i\} \\
& (t - d_i(t) + T_i)u_i - (C_i - e_{i,j}(t)) \\
= & \ \{\text{because } T_i \cdot u_i = C_i\} \\
& (t - d_i(t))u_i + e_{i,j}(t)
\end{aligned}$$

By (3.8) and the above expression for $\mathsf{lag}(\tau_i, t, \mathcal{S})$, we have

$$(t - d_i(t))u_i < \mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - d_i(t))u_i + C_{i,j}.$$

By the definition of the VPP task mode, $C_{i,j} \leq C_i$ for any j. Thus,

$$(t - d_i(t))u_i < \mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - d_i(t))u_i + C_i. \tag{3.11}$$

Rearranging the terms in (3.11) yields (3.7). □

Corollary 3.1. *If $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq L$ for all t, then the tardiness of task τ_i is at most L/u_i.*

Proof. Suppose that

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq L \tag{3.12}$$

[6]Without Assumption (A), $\tau_{i,j}$ might not execute "continuously," so only "$\leq$" can be claimed in (3.10).

holds but τ_i has tardiness exceeding L/u_i. Then, there exists a job $\tau_{i,j}$ that is still pending at some time $t \geq d_{i,j}$ where

$$t - d_{i,j} > L/u_i. \tag{3.13}$$

Because $\tau_{i,j}$ is pending at time t, τ_i is a pending task at time t and its ready job at time t cannot be a job released later than $\tau_{i,j}$. Thus, τ_i's effective deadline at t satisfies $d_i(t) \leq d_{i,j}$. Therefore,

$$\begin{aligned} t - d_i(t) &\geq t - d_{i,j} \\ &> \{\text{by (3.13)}\} \\ &\quad L/u_i \\ &\geq \{\text{by (3.12)}\} \\ &\quad \mathsf{lag}(\tau_i, t, \mathcal{S})/u_i. \end{aligned}$$

That is, $t - \mathsf{lag}(\tau_i, t, \mathcal{S})/u_i > d_i(t)$, which contradicts Lemma 3.2. □

Recall that if $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is negative, then τ_i is over-allocated in schedule $\mathcal{S}$ compared to schedule $\mathcal{I}$. However, the actual schedule $\mathcal{S}$ cannot execute jobs that are not released and therefore can never get more than a full job "ahead" of $\mathcal{I}$. Thus, we have the following trivial lower bound[7] on $\mathsf{lag}(\tau_i, t, \mathcal{S})$, which we state without proof.

Lemma 3.3. $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq -C_{\max}$.

The following lemma uses the relationship between effective deadlines and lag values established in Lemma 3.2 to obtain a sufficient lag-based condition for one task to have an earlier effective deadline than another.

Lemma 3.4. *If tasks τ_i and τ_k are both pending at time t, and if*

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq \frac{u_i}{u_k} \cdot \mathsf{lag}(\tau_k, t, \mathcal{S}) + C_i \tag{3.14}$$

holds, then $d_i(t) < d_k(t)$.

[7] A tighter bound is possible, but this simple bound is sufficient for our purposes.

Proof.

$$
\begin{aligned}
d_i(t) \leq\ & \{\text{by Lemma 3.2}\} \\
& t - \frac{\mathsf{lag}(\tau_i, t, \mathcal{S})}{u_i} + T_i \\
\leq\ & \{\text{by (3.14)}\} \\
& t - \frac{\frac{u_i}{u_k} \cdot \mathsf{lag}(\tau_k, t, \mathcal{S}) + C_i}{u_i} + T_i \\
=\ & \{\text{canceling } u_i \text{ and using } C_i/u_i = T_i\} \\
& t - \frac{\mathsf{lag}(\tau_k, t, \mathcal{S})}{u_k} \\
<\ & \{\text{by Lemma 3.2}\} \\
& d_k(t)
\end{aligned}
$$

The lemma follows. □

Corollary 3.2. *If tasks τ_i and τ_k are both pending at time t and if $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq \rho \cdot \mathsf{lag}(\tau_k, t, \mathcal{S}) + C_{\max}$ holds, where $\rho = u_1/u_n$, then $d_i(t) < d_k(t)$.*

Proof. Because tasks are indexed from highest utilization to lowest, $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq \rho \cdot \mathsf{lag}(\tau_k, t, \mathcal{S}) + C_{\max} = \frac{u_1}{u_n} \cdot \mathsf{lag}(\tau_k, t, \mathcal{S}) + C_{\max} \geq \frac{u_i}{u_k} \cdot \mathsf{lag}(\tau_k, t, \mathcal{S}) + C_i$. By Lemma 3.4, the corollary follows. □

Proof strategies for deriving tardiness bounds. Given the relationships established above between lag values and deadlines, we are now ready to derive tardiness bounds. According to Corollary 3.1, lag bounds directly imply tardiness bounds. Thus, one natural strategy is to attempt to derive n individual lag bounds, one per task. However, we were unable to make this strategy work. Intuitively, this is because, in deriving n individual per-task lag bounds, we must consider how all tasks interact as they are scheduled together. When doing this, it is difficult to avoid a case explosion that causes the entire proof to collapse. In particular, (3.5) and (3.6) must ultimately be exploited in the proof. Every attempt we made in deriving per-task lag bounds resulted in a case explosion that was so unwieldy, we could not discern how (3.5) and (3.6) could possibly factor into the proof.

Notice that (3.5) simply requires that the platform is not over-utilized, which is something required in reasoning about identical platforms as well. The constraints in (3.6), however, are unique to the uniform case.

Observe that these constraints reference the sum of the k largest utilizations and speeds. Accordingly, we switched from working on proof strategies that focus on per-task lag bounds to one that focuses on the sum of the k largest lag values.

Our proof strategy, formally explained. In order to describe this proof strategy more formally, we let $\hat{\tau}_\ell(t)$ denote the task that has the ℓ^{th} largest lag at time instant t, with ties broken arbitrarily. We also denote the ℓ^{th} largest lag at time instant t as $\mathsf{Lag}_l(t)$, *i.e.*, $\mathsf{Lag}_l(t) = \mathsf{lag}(\hat{\tau}_l(t), t, \mathcal{S})$. Furthermore, we let $\mathcal{T}_\ell(t)$ denote the set of tasks corresponding to $\mathsf{Lag}_1(t), \mathsf{Lag}_2(t), \ldots, \mathsf{Lag}_\ell(t)$, *i.e.*, $\mathcal{T}_\ell(t) = \{\hat{\tau}_1(t), \hat{\tau}_2(t), \ldots, \hat{\tau}_\ell(t)\}$.

To derive tardiness bounds, we show that the following $m+1$ inequalities, $(\mathrm{B}_1), \ldots, (\mathrm{B}_m)$, and (B_n), hold at any time t.

Inequality Set (B):

$$\mathsf{Lag}_1(t) \le \beta_1 \tag{B_1}$$

$$\mathsf{Lag}_1(t) + \mathsf{Lag}_2(t) \le \beta_2 \tag{B_2}$$

$$\mathsf{Lag}_1(t) + \mathsf{Lag}_2(t) + \mathsf{Lag}_3(t) \le \beta_3 \tag{B_3}$$

$$\vdots \qquad\qquad \vdots$$

$$\mathsf{Lag}_1(t) + \mathsf{Lag}_2(t) + \cdots + \mathsf{Lag}_k(t) \le \beta_k \tag{B_k}$$

$$\vdots \qquad\qquad \vdots$$

$$\mathsf{Lag}_1(t) + \mathsf{Lag}_2(t) + \cdots + \mathsf{Lag}_m(t) \le \beta_m \tag{B_m}$$

$$\mathsf{Lag}_1(t) + \mathsf{Lag}_2(t) + \cdots + \mathsf{Lag}_n(t) \le \beta_n \tag{B_n}$$

If all constraints in the inequality set (B) hold at all time instants t, where $\beta_1, \beta_2, \ldots, \beta_m$, and β_n are constants (which will depend on task-set parameters), then, by the definition of $\mathsf{Lag}_1(t)$, β_1 is an upper bound on $\mathsf{lag}(\tau_i, t, \mathcal{S})$ for any i and for any t. Given such an upper bound, by Corollary 3.1, tardiness bounds will follow.

In order to prove that the constraints in (B) hold at all time instants t, we must carefully define $\beta_1, \beta_2, \ldots, \beta_m$, and β_n. They are defined as follows.

$$\beta_1 = x_1 \tag{X_1}$$

$$\beta_2 = \beta_1 + x_2 \tag{X_2}$$

$$\beta_3 = \beta_2 + x_3 \tag{X_3}$$

$$\vdots \qquad\qquad \vdots$$

$$\beta_k = \beta_{k-1} + x_k \tag{X_k}$$

$$\vdots \qquad\qquad \vdots$$

$$\beta_m = \beta_{m-1} + x_m \tag{X_m}$$

$$\beta_n = \beta_m + x_n \tag{X_n}$$

where

$$x_n = -(n-m-1)\cdot C_{\max} \tag{Y_1}$$

$$x_m = (n-m+1)\cdot C_{\max} \tag{Y_2}$$

$$x_i = \rho \cdot x_{i+1} + C_{\max}, \quad \text{for } i = m-1, m-2, \ldots, 1 \tag{Y_3}$$

Note that, in (Y_3), $\rho = u_1/u_n$.

To see that $\beta_1, \beta_2, \ldots, \beta_m$, and β_n are well-defined, observe that x_n and x_m can be directly calculated by (Y_1) and (Y_2) for any given task set. Then, $x_{m-1}, x_{m-2}, \ldots, x_1$ can be calculated inductively by (Y_3). Finally, $\beta_1, \beta_2, \ldots, \beta_m$, and β_n can be calculated by $(X_1), \ldots, (X_m)$, and (X_n).

Formal derivation of tardiness bounds. Having set up our proof strategy, we next present a critical mathematical property of the lag and Lag functions when they are viewed as a function of t.

Property 3.1. *For a given task τ_i and a given schedule $\mathcal{S}$, $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is a continuous function of t. For a given schedule $\mathcal{S}$, $\mathsf{Lag}_\ell(t)$ is a continuous function of t for each ℓ.*

Proof. $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is a continuous function of t because, by (3.3), $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\mathcal{I}, \tau_i, 0, t) - \mathsf{A}(\mathcal{S}, \tau_i, 0, t)$, and $\mathsf{A}(\mathcal{I}, \tau_i, 0, t)$ and $\mathsf{A}(\mathcal{S}, \tau_i, 0, t)$ are both (clearly) continuous functions of t. Furthermore, since taking the maximum value of a set of continuous functions is also a continuous function, $\mathsf{Lag}_1(t)$ is a continuous function of t. For similar reasons, $\mathsf{Lag}_2(t)$, $\mathsf{Lag}_3(t)$, ..., $\mathsf{Lag}_n(t)$ are all continuous functions of t as well. □

We are now ready to prove our main theorem.

Theorem 3.3. *At every time instant $t \geq 0$, each inequality in the set (B) holds.*

Proof. Suppose, to the contrary, that the statement of the theorem is not true, and let t_c denote the first time instant such that *any* inequality in (B) is false. We show that the existence of t_c leads to a contradiction.

Claim 3.3. $t_c > 0$.

Proof. It follows by induction using (Y_2) and (Y_3) (and our assumption from Section 3.1 that $n \geq m$) that $x_i > 0$ for $i = 1, 2, \ldots, m$. By induction again, this time using $(\mathrm{X}_1), \ldots, (\mathrm{X}_m)$, it further follows that $\beta_i > 0$ for $i = 1, 2, \ldots, m$. Finally, by (X_m), (X_n), (Y_1), and (Y_2), $\beta_n = \beta_{m-1} + 2C_{\max} > 0$. Thus, because $\mathsf{Lag}_i(0) = 0$ holds for all i, all of the inequalities in (B) are true at time 0, implying that $t_c > 0$. □

Let $t_c^- = t_c - \varepsilon$, where $\varepsilon \to 0^+$.[8] By Claim 3.3, $t_c^- \geq 0$, *i.e.*, t_c^- is well-defined. Because t_c is the *first* time instant at which *any* inequality in (B) is false, *all* such inequalities hold prior to t_c, including at time t_c^-. Also, because the length of the interval $[t_c^-, t_c)$ is arbitrarily small, a task scheduled on a processor at time t_c^- will be continuously scheduled within $[t_c^-, t_c)$.

We call an inequality in (B) *critical* if and only if it is false at time t_c. If (B_k) is critical, then $\mathsf{Lag}_1(t_c^-) + \cdots + \mathsf{Lag}_k(t_c^-) = \beta_k$. This is because (B_k) holds for any time instant before t_c but is falsified at t_c and the left-hand-side of (B_k) is a continuous function of t, by Property 3.1.[9] We now consider two cases, which depend on which inequalities are critical.

Case 1: (B_n) is critical. In this case,

$$\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-) = \beta_n. \tag{3.15}$$

Therefore,

[8] ε does not have to be infinitely close to 0. Instead, it only needs to be a sufficiently small positive constant. However, the criteria for "sufficiently small" are rather tedious, so we merely define $\varepsilon \to 0^+$ here for simplicity. Whenever ε is used, we will further elaborate on its definition in that context.

[9] If $\mathsf{Lag}_1(t_c^-) + \cdots + \mathsf{Lag}_k(t_c^-) < \beta_k$, then a time $t \in [t_c^-, t_c)$ must exist such that $\mathsf{Lag}_1(t) + \cdots + \mathsf{Lag}_k(t) = \beta_k$. Therefore, a smaller ε could have been selected so that $t_c^- = t$.

$$
\begin{aligned}
& \mathsf{Lag}_m(t_c^-) + \mathsf{Lag}_{m+1}(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-) \\
= \; & \{\text{by (3.15)}\} \\
& \beta_n - (\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_{m-1}(t_c^-)) \\
\geq \; & \{\text{because } (\mathrm{B}_{m-1}) \text{ holds at time } t_c^-\} \\
& \beta_n - \beta_{m-1} \\
= \; & \{\text{by } (\mathrm{X}_m), (\mathrm{X}_n), (\mathrm{Y}_1), \text{ and } (\mathrm{Y}_2)\} \\
& 2C_{\max}. \qquad (3.16)
\end{aligned}
$$

Furthermore, by definition, $\mathsf{Lag}_m(t_c^-) \geq \mathsf{Lag}_{m+1}(t_c^-) \geq \cdots \geq \mathsf{Lag}_n(t_c^-)$, so $\mathsf{Lag}_m(t_c^-)$ is at least the average of these $n-m+1$ values. Therefore,

$$
\begin{aligned}
\mathsf{Lag}_m(t_c^-) \geq \; & \frac{\mathsf{Lag}_m(t_c^-) + \mathsf{Lag}_{m+1}(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-)}{n-m+1} \\
\geq \; & \{\text{by (3.16)}\} \\
& \frac{2C_{\max}}{n-m+1} \\
> \; & \{\text{because } C_{\max} > 0 \text{ and } n \geq m\} \\
& 0. \qquad (3.17)
\end{aligned}
$$

Because $\mathsf{Lag}_m(t_c^-)$ denotes the m^{th} largest lag at time t_c^-, (3.17) implies that, at time t_c^-, at least m tasks have positive lag. Thus, by Lemma 3.1, at least m tasks pending at time t_c^-. Therefore, all of the m processors are busy during the time interval $[t_c^-, t_c)$. Thus, by (3.5), the total lag in the system does not increase during the interval $[t_c^-, t_c)$. That is, $\mathsf{Lag}_1(t_c) + \mathsf{Lag}_2(t_c) + \cdots + \mathsf{Lag}_n(t_c) = \sum_{i=1}^n \mathsf{lag}(\tau_i, t_c, \mathcal{S}) \leq \sum_{i=1}^n \mathsf{lag}(\tau_i, t_c^-, \mathcal{S}) = \mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-)$, which by (3.15), implies

$$
\mathsf{Lag}_1(t_c) + \mathsf{Lag}_2(t_c) + \cdots + \mathsf{Lag}_n(t_c) \leq \beta_n.
$$

This contradicts the assumption of Case 1 that (B_n) is critical.

Case 2: (B_k) is critical for some k such that $1 \leq k \leq m$. In this case,

$$\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_k(t_c^-) = \beta_k. \tag{3.18}$$

Our proof for Case 2 utilizes a number of claims, which we prove in turn.

Claim 3.4. $\mathsf{Lag}_k(t_c^-) \geq x_k$.

Proof. If $k = 1$, then by (3.18), $\mathsf{Lag}_1(t_c^-) = \beta_1$. Also, by ($X_1$), $\beta_1 = x_1$, from which $\mathsf{Lag}_1(t_c^-) \geq x_1$ follows. The remaining possibility, $2 \leq k \leq m$, is addressed as follows.

$$\begin{aligned}
\mathsf{Lag}_k(t_c^-) = \; & \{\text{by (3.18)}\} \\
& \beta_k - (\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_{k-1}(t_c^-)) \\
\geq \; & \{\text{since } (B_{k-1}) \text{ holds at time } t_c^-\} \\
& \beta_k - \beta_{k-1} \\
= \; & \{\text{by } (X_k)\} \\
& x_k
\end{aligned}$$

□

Claim 3.5. *If $k \leq m-1$, then $\mathsf{Lag}_{k+1}(t_c^-) \leq x_{k+1}$.*

Proof. Because $k \leq m-1$ and (by assumption) $n \geq m$, $\mathsf{Lag}_{k+1}(t_c^-)$ is well-defined. The claim is established by the following reasoning.

$$\begin{aligned}
\mathsf{Lag}_{k+1}(t_c^-) \leq \; & \{\text{because } (B_{k+1}) \text{ holds at time } t_c^-\} \\
& \beta_{k+1} - (\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_k(t_c^-)) \\
= \; & \{\text{by (3.18)}\} \\
& \beta_{k+1} - \beta_k \\
= \; & \{\text{by } (X_{k+1})\}
\end{aligned}$$

x_{k+1}

□

Claim 3.6. *If $k = m$ and $n > m$, then $\mathsf{Lag}_{k+1}(t_c^-) \leq 0$.*

Proof. $k = m$ implies that (3.18) can be re-written as

$$\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_m(t_c^-) = \beta_m. \tag{3.19}$$

Also, because (B_n) holds at time t_c^-,

$$\mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-) \leq \beta_n. \tag{3.20}$$

Therefore, given $n > m$ (from the statement of the claim), by (3.19) and (3.20), we have

$$\mathsf{Lag}_{m+1}(t_c^-) + \mathsf{Lag}_{m+2}(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-) \leq \beta_n - \beta_m. \tag{3.21}$$

Therefore,

$$\begin{aligned}
\mathsf{Lag}_{m+1}(t_c^-) \leq\ & \{\text{by (3.21)}\} \\
& \beta_n - \beta_m - (\mathsf{Lag}_{m+2}(t_c^-) + \mathsf{Lag}_{m+3}(t_c^-) + \cdots + \mathsf{Lag}_n(t_c^-)) \\
\leq\ & \{\text{by Lemma 3.3}\} \\
& \beta_n - \beta_m - ((-C_{\max}) \cdot (n - m - 1)) \\
=\ & \{\text{by } (\mathrm{X}_n) \text{ and } (\mathrm{Y}_1)\} \\
& -(n - m - 1)C_{\max} + (n - m - 1)C_{\max} \\
=\ & \{\text{canceling}\} \\
& 0.
\end{aligned} \tag{3.22}$$

Because $k = m$, the claim follows from (3.22). □

Claim 3.7. *If $k \leq m-1$ or $n > m$, then $\mathsf{Lag}_k(t_c^-) \geq \rho \cdot \mathsf{Lag}_{k+1}(t_c^-) + C_{\max}$.*

Proof. If $k \leq m-1$, then

$$\begin{aligned}
\mathsf{Lag}_k(t_c^-) \geq\ & \{\text{by Claim 3.4}\} \\
& x_k \\
=\ & \{\text{by } (Y_3)\} \\
& \rho \cdot x_{k+1} + C_{\max} \\
\geq\ & \{\text{by Claim 3.5}\} \\
& \rho \cdot \mathsf{Lag}_{k+1}(t_c^-) + C_{\max}. \qquad (3.23)
\end{aligned}$$

If $k = m$ (recall that $k \leq m$ by the specification of Case 2) and $n > m$, then by Claim 3.6,

$$\mathsf{Lag}_{m+1}(t_c^-) \leq 0. \qquad (3.24)$$

Furthermore,

$$\begin{aligned}
\mathsf{Lag}_m(t_c^-) \geq\ & \{\text{by Claim 3.4}\} \\
& x_m \\
=\ & \{\text{by } (Y_2)\} \\
& (n-m+1)C_{\max} \\
>\ & \{\text{because } C_{\max} > 0 \text{ and } n > m\} \\
& C_{max} \\
\geq\ & \{\text{by (3.24), and because } \rho = u_1/u_n > 0\} \\
& \rho \cdot \mathsf{Lag}_{m+1}(t_c^-) + C_{\max}. \qquad (3.25)
\end{aligned}$$

Thus, by (3.23) and (3.25) the claim follows. □

In considering the next claim, recall the following: at time t, $\mathsf{Lag}_\ell(t)$ denotes the ℓ^{th} largest lag among all n tasks and $\mathcal{T}_\ell(t)$ denotes the set of ℓ tasks with largest lag values.

Claim 3.8. *The k tasks in $\mathcal{T}_k(t_c^-)$ are scheduled on the k fastest processors within $[t_c^-, t_c)$.*

Proof. Because $\rho = u_1/u_n \geq 1$ and (by assumption) $n \geq m$, by (Y_2) and (Y_3), it can be shown that $x_k \geq C_{\max}$ for $1 \leq k \leq m$. Therefore, by Claim 3.4, we have $\mathsf{Lag}_k(t_c^-) \geq x_k \geq C_{\max} > 0$. By Lemma 3.1, this implies that each of the k tasks in $\mathcal{T}_k(t_c^-)$ is pending at time t_c^-. Therefore, by Policy (G), it suffices to prove that the k tasks in $\mathcal{T}_k(t_c^-)$ have the k earliest effective deadlines at time t_c^- (with no tie with the $(k+1)^{st}$ earliest effective deadline).[10] If $k = m$ and $n = m$, then there are k tasks in total in the system. In this case, the k pending tasks in $\mathcal{T}_k(t_c^-)$ clearly have the k earliest effective deadlines at time t_c^-.

In the rest of the proof, we consider the remaining possibility, *i.e.*, $k \leq m-1$ or $n > m$. By Claim 3.7,

$$\mathsf{Lag}_k(t_c^-) \geq \rho \cdot \mathsf{Lag}_{k+1}(t_c^-) + C_{\max}. \tag{3.26}$$

By the definition of Lag, (3.26) implies that for any i and j such that $1 \leq i \leq k$ and $k+1 \leq j \leq n$, we have $\mathsf{Lag}_i(t_c^-) \geq \rho \cdot \mathsf{Lag}_j(t_c^-) + C_{\max}$. This implies that, for any task $\tau_p \in \mathcal{T}_k(t_c^-)$ and any task $\tau_q \notin \mathcal{T}_k(t_c^-)$, $\mathsf{lag}(\tau_p, t_c^-, \mathcal{S}) \geq \rho \cdot \mathsf{lag}(\tau_q, t_c^-, \mathcal{S}) + C_{\max}$. Therefore, if a task $\tau_q \notin \mathcal{T}_k(t_c^-)$ is pending at time t_c^-, then by Lemma 3.4, its effective deadline is strictly greater than the effective deadline of any task $\tau_p \in \mathcal{T}_k(t_c^-)$; if τ_q is not pending at time t_c^-, then it has no pending jobs and no effective deadline by definition. Therefore, the k tasks in $\mathcal{T}_k(t_c^-)$ have the k earliest effective deadlines at time t_c^-. The claim follows. □

Claim 3.9. $\mathcal{T}_k(t_c^-) = \mathcal{T}_k(t_c)$.

Proof. If $k = m$ and $n = m$, then there are k tasks in total in the system, so the claim clearly holds. Therefore, in the rest of the proof, we assume $k \leq m-1$ or $n > m$, which implies that either $k \leq m-1$ holds or $k = m$ and $n > m$ hold, by the specification of Case 2. If $k \leq m-1$, then

$$\mathsf{Lag}_k(t_c^-) \geq \ \{\text{by Claim 3.4}\}$$

[10] Note that ε can be selected to be small enough to ensure that no scheduling event happens within the interval $[t_c^-, t_c)$, including job completions. Therefore, any task scheduled at time t_c^- will continuously execute during time interval $[t_c^-, t_c)$ on the same processor.

$$
\begin{aligned}
& x_k \\
= \; & \{\text{by } (\mathrm{Y}_3)\} \\
& \rho \cdot x_{k+1} + C_{max} \\
\geq \; & \{\text{because } \rho = u_1/u_n \geq 1 \text{ and } x_{k+1} \geq 0\} \\
& x_{k+1} + C_{\max} \\
\geq \; & \{\text{by Claim 3.5}\} \\
& \mathsf{Lag}_{k+1}(t_c^-) + C_{\max}. \qquad (3.27)
\end{aligned}
$$

If $k = m$ and $n > m$, then

$$
\begin{aligned}
\mathsf{Lag}_k(t_c^-) \geq \; & \{\text{by Claim 3.4 and because } k = m\} \\
& x_m \\
= \; & \{\text{by } (\mathrm{Y}_2)\} \\
& (n-m+1)C_{max} \\
> \; & \{\text{because } n > m\} \\
& C_{\max} \\
\geq \; & \{\text{because } \mathsf{Lag}_{k+1}(t_c^-) \leq 0, \text{ by Claim 3.6}\} \\
& \mathsf{Lag}_{k+1}(t_c^-) + C_{\max}. \qquad (3.28)
\end{aligned}
$$

Thus, for $k \leq m-1$ or $n > m$, by (3.27) and (3.28), we have

$$
\mathsf{Lag}_k(t_c^-) - \mathsf{Lag}_{k+1}(t_c^-) \geq C_{\max}. \qquad (3.29)
$$

By the definition of Lag, (3.29) implies that for any i and j such that $1 \leq i \leq k$ and $k+1 \leq j \leq n$, we have $\mathsf{Lag}_i(t_c^-) - \mathsf{Lag}_j(t_c^-) \geq C_{\max}$. This implies that, for any task $\tau_p \in \mathcal{T}_k(t_c^-)$ and any task $\tau_q \notin \mathcal{T}_k(t_c^-)$, we have

$$
\mathsf{lag}(\tau_p, t_c^-, \mathcal{S}) - \mathsf{lag}(\tau_q, t_c^-, \mathcal{S}) \geq C_{\max}. \qquad (3.30)
$$

Therefore,

$$
\begin{aligned}
& \mathsf{lag}(\tau_p, t_c, \mathcal{S}) - \mathsf{lag}(\tau_q, t_c, \mathcal{S}) \\
= \; & \{\text{by (3.4)}\} \\
& \mathsf{lag}(\tau_p, t_c^-, \mathcal{S}) + \mathsf{A}(\mathcal{I}, \tau_p, t_c^-, t_c) - \mathsf{A}(\mathcal{S}, \tau_p, t_c^-, t_c) \\
& \quad - \mathsf{lag}(\tau_q, t_c^-, \mathcal{S}) - \mathsf{A}(\mathcal{I}, \tau_q, t_c^-, t_c) + \mathsf{A}(\mathcal{S}, \tau_q, t_c^-, t_c) \\
\geq \; & \{\text{since } 0 \leq \mathsf{A}(\mathcal{I}, \tau_i, t_1, t_2) \leq u_1 \cdot (t_2 - t_1) \text{ and } 0 \leq \mathsf{A}(\mathcal{S}, \tau_i, t_1, t_2) \leq s_1 \cdot (t_2 - t_1)\} \\
& \mathsf{lag}(\tau_p, t_c^-, \mathcal{S}) + 0 - \varepsilon \cdot s_1 - \mathsf{lag}(\tau_q, t_c^-, \mathcal{S}) - \varepsilon \cdot u_1 + 0 \\
\geq \; & \{\text{by (3.30) and rearranging}\} \\
& C_{\max} - \varepsilon \cdot (s_1 + u_1) \\
> \; & \{\text{because } C_{\max} > 0 \text{ and } \varepsilon < C_{\max}/(s_1 + u_1)\}^{11} \\
& 0.
\end{aligned}
$$

Thus, at time t_c, any task in $\mathcal{T}_k(t_c^-)$ has a strictly greater lag than any task not in $\mathcal{T}_k(t_c^-)$. This implies that $\mathcal{T}_k(t_c)$ and $\mathcal{T}_k(t_c^-)$ consist of the same set of tasks. □

By Claim 3.8 and Claim 3.9, the k tasks in $\mathcal{T}_k(t_c)$ are continuously scheduled on the k fastest processor during time interval $[t_c^-, t_c)$, so

$$\sum_{\tau_i \in \mathcal{T}_k(t_c)} \mathsf{A}(\mathcal{S}, \tau_i, t_c^-, t_c) = S_k \cdot \varepsilon. \tag{3.31}$$

Also, by (3.1),

$$\sum_{\tau_i \in \mathcal{T}_k(t_c)} \mathsf{A}(\mathcal{I}, \tau_i, t_c^-, t_c) \leq \sum_{\tau_i \in \mathcal{T}_k(t_c)} u_i \cdot \varepsilon \leq U_k \cdot \varepsilon. \tag{3.32}$$

Therefore,

$$
\begin{aligned}
& \mathsf{Lag}_1(t_c) + \mathsf{Lag}_2(t_c) + \cdots + \mathsf{Lag}_k(t_c) \\
= \; & \{\text{by Claim 3.9 and by (3.4)}\} \\
& \mathsf{Lag}_1(t_c^-) + \mathsf{Lag}_2(t_c^-) + \cdots + \mathsf{Lag}_k(t_c^-) + \sum_{\tau_i \in \mathcal{T}_k(t_c)} \mathsf{A}(\mathcal{I}, \tau_i, t_c^-, t_c) - \sum_{\tau_i \in \mathcal{T}_k(t_c)} \mathsf{A}(\mathcal{S}, \tau_i, t_c^-, t_c)
\end{aligned}
$$

[11] ε can be selected small enough to ensure $\varepsilon < C_{\max}/(s_1 + u_1)$.

$\leq$ {by (3.31) and (3.32), and because (B_k) holds at time t_c^-}

$\beta_k + (U_k - S_k) \cdot \varepsilon$

$\leq$ {by (3.5) and (3.6); note that $U_m \leq U_n$}

β_k.

This contradicts the assumption of Case 2 that (B_k) is critical.

Finishing up. We have shown that both Case 1 and Case 2 lead to a contradiction. That is, none of the conditions $(\mathrm{B}_1), \ldots, (\mathrm{B}_m)$, or (B_n) is critical at t_c. This contradicts the definition of t_c as the first time instant at which some inequality in (B) is false, *i.e.*, such a t_c does not exist. The theorem follows. □

Using Theorem 3.3, we can easily derive a tardiness bound for every task as follows.

Theorem 3.4. *In $\mathcal{S}$, the tardiness Δ_i of task τ_i is bounded as follows.*

$$\textit{If } \rho = 1, \textit{ then } \Delta_i \leq \frac{n \cdot C_{\max}}{u_i};$$

$$\textit{if } \rho > 1, \textit{ then } \Delta_i \leq \frac{\rho^{m-1} \cdot (n-m+1)C_{\max} + \frac{\rho^{m-1}-1}{\rho-1} \cdot C_{\max}}{u_i}.$$

Proof. By Theorem 3.3, $\mathsf{Lag}_1(t) \leq \beta_1$ holds at every time instant t. By the definition of $\mathsf{Lag}_1(t)$, this implies that, for each i, $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq \beta_1$ holds for all t. Thus, by Corollary 3.1, task τ_i has a tardiness bound of β_1/u_i. By (X_1), $\beta_1 = x_1$, so to complete the proof, we merely need to calculate x_1.

If $\rho = 1$ (*i.e.*, $u_1/u_n = 1$, which implies that every task has the same utilization), then by (Y_3), $x_1 = x_m + (m-1)C_{\max}$. Thus, by (Y_2), we have $x_1 = n \cdot C_{\max}$.

If $\rho > 1$, then rearranging (Y_3) results in

$$x_i + \frac{C_{\max}}{\rho - 1} = \rho \left(x_{i+1} + \frac{C_{\max}}{\rho - 1} \right).$$

By iterating this recurrence, we have

$$x_1 + \frac{C_{\max}}{\rho - 1} = \rho^{m-1} \cdot \left(x_m + \frac{C_{\max}}{\rho - 1} \right).$$

Finally, applying (Y_2) yields

$$x_1 = \rho^{m-1} \cdot (n-m+1)C_{\max} + \frac{\rho^{m-1}-1}{\rho-1} \cdot C_{\max}.$$

The theorem follows. □

Discussion. Theorem 3.4 provides a tardiness bound for any task in a VPP task system that satisfies (3.5) and (3.6). As shown in Section 3.5.1, any sporadic task is a special case of a VPP task, and therefore the tardiness bound in Theorem 3.4 also applies to a sporadic task system that satisfies (3.5) and (3.6). Furthermore, as shown in Section 3.2, (3.5) and (3.6) are a necessary and sufficient SRT-feasibility condition for sporadic task systems on a uniform multiprocessor. Thus, Theorem 3.4 leads to the conclusion that preemptive G-EDF is SRT-optimal for sporadic task systems on uniform multiprocessors.

In addition, Theorem 3.4 also shows that (3.5) and (3.6) are a sufficient feasibility condition for VPP task systems on a uniform multiprocessor, and the necessity can be shown by the same reasoning for sporadic task systems in Section 3.2. Thus, (3.5) and (3.6) are a necessary and sufficient SRT-feasibility condition and preemptive G-EDF is SRT-optimal also for VPP task systems on a uniform multiprocessor.

3.6 Chapter Summary

In this chapter, we have considered the problem of using G-EDF to schedule sporadic SRT tasks on a uniform multiprocessor. We have shown that non-preemptive G-EDF is not SRT-optimal for uniform multiprocessors by providing a counterexample, and this negative result in fact applies to any work-conserving non-preemptive scheduling algorithm. On the other hand, we have proved that preemptive G-EDF is indeed SRT-optimal for uniform multiprocessors by deriving tardiness bounds for an arbitrary SRT-feasible system, and this result applies to the VPP task model, which is a more general task model than the sporadic task model.

CHAPTER 4: SEMI-PARTITIONED SCHEDULING ON UNIFORM PLATFORMS[1]

In this chapter, we continue to study the problem of scheduling a set of sporadic tasks on a uniform multiprocessor. In contrast to the global scheduling approach studied in the prior chapter, we will focus on *semi-partitioned* scheduling in this chapter. Traditionally, a multiprocessor scheduling algorithm may follow a global approach, in which any migration is allowed, or a partitioned approach, in which no migration is allowed. As a hybrid of these two approaches, a *semi-partitioned* scheduling algorithm allows only a limited number of tasks to migration but requires all remaining ones to be fixed to processors.

We will present two semi-partitioned scheduling algorithms designed for uniform multiprocessors in this chapter, namely EDF-sh (earliest-deadline-first-based semi-partitioned scheduler for uniform heterogeneous multiprocessors) and EDF-tu (earliest-deadline-first-based tunable scheduler for uniform platforms). In addition to only allowing a limited number of tasks to migrate, EDF-sh further requires these tasks to migrate at job boundaries only, at the cost of supporting SRT tasks only and being not SRT-optimal. In contrast, EDF-tu is always SRT-optimal, and may be HRT-optimal if a tunable parameter, called *frame size*, divides all task periods. For SRT tasks, any frame size can be selected, and tardiness is upper bounded by the value of the frame size as long as the system is feasible. However, a smaller frame size potentially leads to more frequent preemptions and migrations.

Organization. In the following sections, we first introduce the common system model for both algorithms (Section 4.1), and then present EDF-sh (Section 4.2) and EDF-tu (Section 4.3) respectively in details.

4.1 System Model

In this chapter, we consider scheduling n sporadic tasks on m processors, where $n \geq m$. Processor p is identified by its speed s_p ($1 \leq p \leq m$, $s_p \in \mathbb{R}$). We also assume implicit deadlines, *i.e.*, each task has a

[1]Contents of this chapter previously appeared in preliminary form in the following papers:

Yang, K. and Anderson, J. (2014b). Soft real-time semi-partitioned scheduling with restricted migrations on uniform heterogeneous multiprocessors. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 215–224.

Yang, K. and Anderson, J. (2015b). An optimal semi-partitioned scheduler for uniform heterogeneous multiprocessors. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 199–210.

relative deadline equal to its period. Thus, a task τ_i can be specified by $\tau_i = (C_i, T_i)$, where C_i is its *worst-case execution requirement* and T_i is its *period*. We define the *utilization* of a task τ_i as

$$u_i = \frac{C_i}{T_i}. \tag{4.1}$$

A *job* is an invocation of a task; the j^{th} job of task τ_i is denoted $\tau_{i,j}$. $r_{i,j}$ is its release time and $d_{i,j}$ is its absolute deadline, where $d_{i,j} = r_{i,j} + T_i$. The *tardiness* of a job $\tau_{i,j}$ that completes at time t_c is defined as $\max\{0, t_c - d_{i,j}\}$, while its *lateness* is $t_c - d_{i,j}$. The two differ only if $\tau_{i,j}$ completes before its deadline, in which case its tardiness is zero but its lateness is negative. The *speed* of a processor refers to the amount of work completed in one time unit when a job is executed on that processor. Moreover, jobs of the same task cannot execute in parallel, *i.e.*, a job can commence execution only when all prior jobs of the same task have finished.

We index the processors in non-increasing-speed order, *i.e.*, $\pi = \{s_1, s_2, \cdots, s_m\}$, where $s_p \geq s_{p+1}$ for $p \in \{1, 2, \cdots, m-1\}$; and we also index the tasks in non-increasing-utilization order, *i.e.*, $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where $u_i \geq u_{i+1}$ for $i \in \{1, 2, \cdots, n-1\}$. We also define $U_k = \sum_{i=1}^{k} u_i$ and $S_k = \sum_{i=1}^{k} s_i$.

Feasibility Condition. As shown in Section 3.2, the SRT-feasibility condition matches the HRT-feasibility condition for scheduling implicit-deadline sporadic tasks on a uniform multiprocessor, and therefore we omit the "SRT" and "HRT" for feasibility in this case. That is, a set of sporadic tasks τ is feasible on a uniform multiprocessor π if and only if

$$U_n \leq S_m \tag{4.2}$$

and

$$U_k \leq S_k \quad \text{for} \quad k = 1, 2, \cdots, m-1. \tag{4.3}$$

Fixed and migrating tasks. Under semi-partitioned scheduling, each task is allocated a non-zero *share* on certain processors such that the total allocated share on each processor does not exceed the processor's capacity and the total allocated share of a task matches its utilization. If a task has non-zero shares on only one (multiple) processor(s), then it is a *fixed* (*migrating*) task.

4.2 EDF-sh

In this section, we present the first algorithm, EDF-sh. Like many other semi-partitioned algorithms, it has an *assignment phase* and an *execution phase*. We first discuss utilization constraints that are required under EDF-sh and describe its two phases (Section 4.2.1), and then prove tardiness bounds for task systems under EDF-sh (Section 4.2.2). Finally, we present an evaluation (Section 4.2.3).

4.2.1 Algorithm EDF-sh

We design EDF-sh by extending EDF-os (Anderson et al., 2016) to uniform multiprocessors.

As a result, EDF-sh inherits most of the advantages of EDF-os, such as:

- Under EDF-sh, every job has bounded tardiness.
- Migrations are boundary-limited.
- The underlying platform can be fully utilized, *i.e.*, U_n can be as large as S_m.

In the tardiness-bound proof for EDF-os, and for EDF-sh here, it is essential that each task executes only on processors that have a speed at least its utilization without overutilizing any processor. Unfortunately, this cannot be ensured for all feasible task systems (recall (4.2) and (4.3)). For example, a task system $\tau = \{(2,1),(2,1)\}$ to be scheduled on $\pi = \{3,1\}$ is feasible, but if we assign jobs of each task only to processors with a speed at least its utilization, then the first processor will be overutilized. Because of this difficulty, we further restrict task utilizations slightly by requiring

$$\sum_{u_i > s_k} u_i \leq \sum_{s_p > s_k} s_p \quad \text{for} \quad k = 1,2,\cdots,m, \tag{4.4}$$

which is a little more restrictive than (4.3). Nevertheless, the total utilization U_n can be as large as the total speed S_m.

Note that (4.4) implies (4.3). Thus, we omit (4.3) and hence let (4.2) and (4.4) be our task system utilization restriction for EDF-sh in Section 4.2.

Similarly to EDF-os, EDF-sh has two phases, an assignment phase and an execution phase. In the assignment phase, we consider tasks in non-increasing-utilization order. When considering a task, we first check the current available capacity of each processor to see if this task can be fixed. If so, we assign this task

Algorithm 1 EDF-sh task assignment phase

initially $\psi_{i,p} = 0$ and $\sigma_p = 0$ for all i and p index tasks in a non-increasing-utilization order index processors in a non-increasing-speed order /* p is the index of the last processor to which a migrating task was assigned (or 1, if no migrating task has been assigned yet). s_p is the first processor for next migrating task if its capacity has not been exhausted yet. */

$p := 1$;

for $i := 1$ **to** n **do**

  /* If task τ_i can be fixed, then we assign it to be fixed task via worst-fit here. */

  Select k that $s_k - \sigma_k$ is maximal;

  if $s_k - \sigma_k \geq u_i$ **then**

    $\psi_{i,k} = u_i$;

    $\sigma_k = \sigma_k + u_i$;

  else

    /* If task τ_i has to migrate, then we assign its shares on processors to exhaust processor capacities in turn from the fastest one to the slowest one. */

    $remaining := u_i$;

    repeat

      $\psi_{i,p} := \min(remaining, (s_p - \sigma_p))$

      $\sigma_p := \sigma_p + \psi_{i,p}$;

      $remaining := remaining - \psi_{i,p}$;

      if $\sigma_p = s_p$ **then**

        $p := p + 1$;

      end

    until $remaining = 0$;

  end

end

to some processor as a fixed task via a bin-packing heuristic. The specific heuristic does not matter in terms of theoretical schedulability; we choose to use worst-fit here. The assignment phase of EDF-sh is defined by the pseudo-code in Algorithm 1, where $\psi_{i,p}$ denotes the share (which potentially can be zero) of task τ_i on processor p and the total share allocation on processor p is denoted as $\sigma_p = \sum_{k=1}^{n} \psi_{k,p}$. Algorithm 1 maintains that no processor is overutilized, *i.e.*, $\sigma_p \leq 1.0$ holds for all p. Also, the total share allocation of a task τ_i matches its utilization, *i.e.*, $\sum_{k=1}^{m} \psi_{i,k} = u_i$

We use $\varphi_{i,p}$ to denote the long-term *fraction* of task τ_i's jobs that execute on processor p. $\varphi_{i,p}$ is commensurate with the share allocated:

$$\varphi_{i,p} = \frac{\psi_{i,p}}{u_i}. \tag{4.5}$$

The set of all fixed tasks on s_p is denoted τ_p^f, and $\sigma_p^f = \sum_{\tau_i \in \tau_p^f} \psi_{i,p}$.

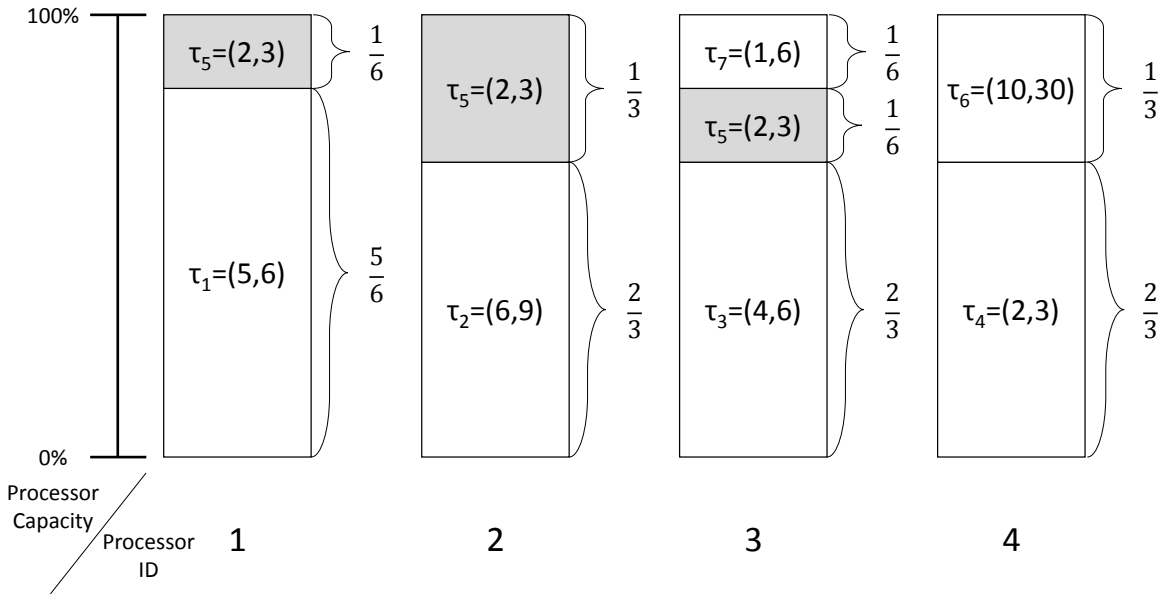


Figure 4.1: EDF-sh task assignment for Example 4.1. This is the same system as in Example 2.2, but EDF-sh has a different assignment from EDF-os.

Example 4.1. To illustrate the difference between the assignment phases of EDF-os and EDF-sh, we revisit the system in Example 2.2. Note that any identical multiprocessor is a uniform one (with $s_p = 1$ for all p), so EDF-sh also works on identical multiprocessors. The assignment of the first five tasks by EDF-sh is exactly the same as that by EDF-os. However, we will attempt to make all remaining tasks fixed as well, and this results in τ_6 being fixed on processor 4 and thereafter τ_7 being fixed on processor 3. That is, EDF-sh will have only one migrating task for this system. Figure 4.1 shows the resulting assignment by EDF-sh. $\Diamond$

Example 4.2. We now give an example of the task assignment phase of EDF-sh for the case where processor speeds are different. In this example, we have a uniform multiprocessor $\pi = \{4,2,2,1\}$, upon which a set of sporadic tasks $\tau = \{(3,1),(11,6),(5,3),(4,3),(1,2),(2,6),(1,3)\}$ will be scheduled. Via the worst-fit heuristic, τ_1, τ_2, and τ_3 are assigned as fixed tasks to s_1, s_2, and s_3, respectively. Thereafter, no single processor has enough capacity to fix τ_4, so τ_4 must migrate. It is assigned non-zero shares on s_1, s_2, and s_3. However, next, τ_5 and τ_6 can be fixed, specifically on s_4. Finally, τ_7 must migrate between s_3 and s_4. The resulting task assignment is depicted in Figure 4.2. For the two migrating tasks, s_3 is the last processor of τ_4, and s_4 is the last processor of τ_7. $\Diamond$

The assignment phase of EDF-sh ensures the following properties.

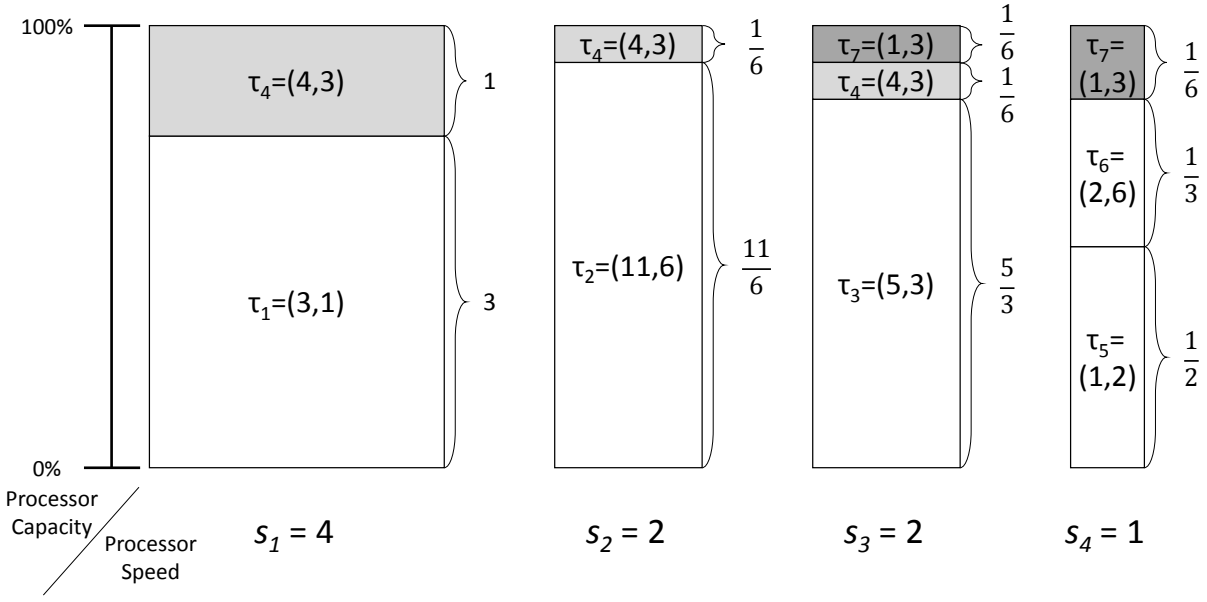


Figure 4.2: EDF-sh task assignment for Example 4.2. The width of each column indicates the processor speed.

Property 4.1. *There are no more than two migrating tasks on s_p. If there are exactly two migrating tasks on s_p, then s_p is the last processor for exactly one of them.*

This property follows from the assignment procedure, and it can be proved by induction.

By Property 4.1, we know that a processor s_p will have at most two migrating tasks, and if exactly two, then they must be of different priorities. Therefore, if there is only one migrating task on a given processor s_p, then we use τ_l to denote that task; if there are two, then we let τ_l (τ_h) denote the migrating task with lower (higher) priority.

Property 4.2. *For any processor s_p, $\sigma_p^f + \psi_{h,p} + \psi_{l,p} \leq s_p$. (If for s_p, τ_h and/or τ_l do not exist, then we just consider $\psi_{h,p}$ and/or $\psi_{l,p}$ to be zero.)*

This property holds because in our assignment phase, we do not overutilize any processor, *i.e.*, we always maintain $\sigma_p \leq s_p$.

Property 4.3. *A task has non-zero shares only on processors that have a speed at least its utilization.*

Proof. This property clearly holds for fixed tasks. We show that it holds for migrating tasks as well by contradiction.

Suppose there is some migrating task that violates this property. Let τ_a be the first migrating task to do so, and let s_q be the first processor such that $s_q < u_a$ where τ_a is assigned a non-zero share. In Algorithm 1, we do not assign a migrating task a non-zero share on a slower processor unless the capacity of every faster one has been exhausted. By the definition of τ_a, Property 4.3 holds for all previously assigned migrating tasks and hence *all* previously assigned tasks, since it trivially holds for any fixed task. Moreover, because tasks are considered in non-increasing-utilization order, every such prior task has a utilization at least u_a and therefore larger than s_q. These facts imply that all prior tasks have been assigned shares only on the first $q-1$ processors, and including τ_a, the total allocated shares of the first a tasks exceeds the capacity of the first $q-1$ processors. Thus, we have

$$\sum_{i=1}^{a} u_i > S_{q-1}. \tag{4.6}$$

Since the processors are indexed in non-increasing-speed order,

$$S_{q-1} \geq \sum_{s_p > s_q} s_p. \tag{4.7}$$

Since $u_a > s_q$ and the tasks are indexed in non-increasing-utilization order, $u_i \geq u_a > s_q$ holds for all $i \leq a$. That is,

$$\sum_{u_i > s_q} u_i \geq \sum_{i=1}^{a} u_i. \tag{4.8}$$

By (4.6), (4.7), and (4.8), we have

$$\sum_{u_i > s_q} u_i > \sum_{s_p > s_q} s_p,$$

which contradicts (4.4). Thus, no such τ_a exists and therefore Property 4.3 holds. □

Property 4.4. *When we assign a migrating task a non-zero share on a processor, there must be at least one fixed task on that processor.*

Proof. Suppose this property is violated for the first time when migrating task τ_i is assigned a non-zero share on processor s_p, *i.e.*, there is no fixed task on s_p. Since τ_i is the first migrating task that violates Property 4.4, no other migrating task is assigned a non-zero share on s_p either. Because no prior task (fixed or migrating) is assigned a non-zero share on s_p and $s_p \geq u_i$ (by Property 4.3), τ_i would be assigned as fixed on s_p, which contradicts our assumption that it is a migrating task. Thus, no such τ_i exists and hence this property holds. □

Property 4.5. *If there are exactly two migrating tasks on s_p, i.e., $\psi_{h,p} > 0$ and $\psi_{l,p} > 0$, then $\psi_{h,p} + u_l < s_p$.*

Proof. By Property 4.4, there must be at least one fixed task τ_a that was assigned to s_p before the two migrating ones are assigned shares on s_p. Since the tasks are considered in non-increasing-utilization order, we have $u_l \leq u_a$. Also, by the definition of σ_p^f, $\sigma_p^f \geq u_a$, so $u_l \leq \sigma_p^f$. Therefore,

$$\begin{aligned} u_l + \psi_{h,p} + \psi_{l,p} &\leq \sigma_p^f + \psi_{h,p} + \psi_{l,p} \\ &\leq \{\text{by Property 4.2}\} \\ &s_p. \end{aligned}$$

Because $\psi_{l,p} > 0$ here, we get $\psi_{h,p} + u_l < s_p$. □

In the execution phase, every fixed task will only release jobs on the processor to which it assigned, whereas jobs of migrating tasks will be assigned to processors in the same way as in EDF-os (Anderson et al., 2016). Therefore, the following property from EDF-os holds for EDF-sh as well.

Property 4.6. *For any k consecutive jobs of a migrating task τ_i, at most $\varphi_{i,p} \cdot k + 2$ of them are assigned to processor s_p.*

Furthermore, the following scheduling rules are applied on each processor in the execution phase.

- Jobs of migrating tasks are statically prioritized over those of fixed tasks.
- Jobs of fixed tasks are prioritized against each other on an EDF basis.
- On a migrating task's *last* processor, its priority is lower than other migrating tasks, but still higher than fixed ones.

4.2.2 Tardiness Bounds

In this section, we prove tardiness bounds for EDF-sh. We consider migrating tasks and fixed task separately in Sections. 4.2.2.1 and 4.2.2.2. Moreover, for migrating tasks, rather than tardiness, we upper bound lateness for each task.

4.2.2.1 Migrating Tasks

In this subsection, we derive lateness bounds for migrating tasks. Since migrating tasks are statically prioritized over fixed ones, we can ignore all fixed tasks when considering migrating ones.

Lemma 4.1. *If the lateness of jobs of task τ_i is upper bounded by Δ_i, then in the time interval $[t_0, t_c)$, the demand from τ_i on processor s_p is less than*

$$\psi_{i,p} \cdot (t_c - t_0) + \psi_{i,p} \cdot (2T_i + \Delta_i) + 2C_i.$$

Proof. Because lateness is upper bounded by Δ_i, the jobs of τ_i released before $t_0 - (T_i + \Delta_i)$ complete their execution by t_0. Therefore, in the time interval $[t_0, t_c)$, the demand from τ_i can only come from its jobs released in $[t_0 - T_i - \Delta_i, t_c)$. τ_i can release at most $\left\lceil \frac{t_c - (t_0 - T_i - \Delta_i)}{T_i} \right\rceil$ jobs in $[t_0 - T_i - \Delta_i, t_c)$. By Property 4.6, at most $\varphi_{i,p} \cdot \left\lceil \frac{t_c - (t_0 - T_i - \Delta_i)}{T_i} \right\rceil + 2$ of them are assigned to processor s_p. Thus, in the time interval $[t_0, t_c)$, the demand from τ_i on processor s_p is at most

$$\left(\varphi_{i,p} \cdot \left\lceil \frac{t_c - (t_0 - T_i - \Delta_i)}{T_i} \right\rceil + 2 \right) \cdot C_i$$

$$< \{\text{since } \lceil x \rceil < x + 1\}$$
$$\left(\varphi_{i,p} \cdot \left(\frac{t_c - (t_0 - T_i - \Delta_i)}{T_i} + 1 \right) + 2 \right) \cdot C_i$$

$$= \{\text{simplifying}\}$$
$$\left(\varphi_{i,p} \cdot \frac{t_c - t_0 + 2T_i + \Delta_i}{T_i} + 2 \right) \cdot C_i$$

$$= \{\text{by (4.1) and (4.5)}\}$$
$$\psi_{i,p} \cdot (t_c - t_0) + \psi_{i,p} \cdot (2T_i + \Delta_i) + 2C_i.$$

The lemma follows. □

According to the following lemma, if we assume that all the jobs of a migrating task are moved to its last processor, then the lateness of these jobs under this assumption upper bounds their lateness in the actual schedule. This analysis device was first used by Anderson et al. (2016), but assuming such jobs are moved to

the task's *first* processor. We must instead consider the *last* processor, because in our case processors may have different speeds. Thus, we must conservatively assume such moves are with respect to a task's slowest processor.

Lemma 4.2. *If we execute all jobs of a migrating task on its last processor rather than the processors where these jobs are actually assigned, then no job of this task will complete its execution earlier. Moreover such job moves do not impact the other migrating task on this processor (if one exits).*

Proof. A migrating task has the highest priority on any processor that is not its last processor. Also, on any non-last processor, a migrating task is executed at a speed at least that of its last processor, since the processors are indexed from fastest to slowest. Thus, the first part of Lemma 4.2 holds.

The second part of Lemma 4.2 follows because on its last processor, a migrating task has statically lower priority than any other migrating task. □

Property 4.3 and Property 4.5 ensure that, with respect to migrating tasks, such job moves will not overutilize the last processor of the considered task.

We now compute a lateness bound for each migrating task assuming its jobs are moved as described above. If a task is the only migrating task on its last processor, then its lateness bound can be computed directly; if a migrating task τ_l shares its last processor with another migrating task τ_h, then its lateness bound depends on the lateness bound of τ_h, which can be computed inductively by the formula in Theorem 4.1. Lemma 4.3 below ensures that the base case exists.

Lemma 4.3. *The migrating task with the largest index does not share its last processor with any other migrating task.*

Proof. Follows directly from Algorithm 1. □

Theorem 4.1. *Consider a migrating task τ_l that has s_p as its last processor. If it shares s_p with some other migrating task τ_h, then τ_h is the only such task (by Property 4.1). Let Δ_h be the lateness bound of τ_h. Then τ_l has lateness at most*

$$\Delta_l \stackrel{def}{=} \begin{cases} \frac{\psi_{h,p}\cdot(2T_h+\Delta_h)+2C_h+C_l}{s_p-\psi_{h,p}} - T_l & \text{if } \tau_h \text{ exists,} \\ \frac{C_l}{s_p} - T_l & \text{otherwise.} \end{cases} \tag{4.9}$$

Proof. By Lemma 4.2, we can upper bound the lateness of all jobs of τ_l by assuming that all such jobs execute on s_p. We make that assumption here.

If τ_l does not share its last processor s_p with any other migrating task, *i.e.*, τ_h does not exist, then τ_l has the highest priority on s_p, and by Property 4.3, $s_p \geq u_l$. Therefore, every job of τ_l completes its execution within $\frac{C_l}{s_p}$ time units of its release. Thus, lateness is upper bounded by $\frac{C_l}{s_p} - T_l$.

In the remainder of the proof, we consider the case where τ_h does exist. In this case, we prove Theorem 4.1 by contradiction.

Interval $[t_0, t_c)$. Let $\tau_{l,j}$ be the first job of τ_l that has lateness exceeding Δ_l and define $t_c \stackrel{\text{def}}{=} d_{l,j} + \Delta_l$. Let t_0 be the latest time instant before t_c such that s_p is idle for migrating tasks, *i.e.*, all jobs of τ_l or τ_h released before t_0 have completed execution by t_0 and a job of τ_l and/or τ_h is released at t_0. t_0 is well defined because if no such time instant exists within $(0, t_c)$, then time 0 must be such a time instant.

Demand from τ_h. By Lemma 4.1, in the time interval $[t_0, t_c)$, the demand from τ_h on s_p is less than

$$\psi_{h,p} \cdot (t_c - t_0) + \psi_{h,p} \cdot (2T_h + \Delta_h) + 2C_h. \tag{4.10}$$

Demand from τ_l. The demand from τ_l on s_p comes from jobs of τ_l released before $r_{l,j}$ and $\tau_{l,j}$ itself. By the definition of t_0, the number of such jobs released before $r_{l,j}$ is at most $\left\lfloor \frac{r_{l,j}-t_0}{T_l} \right\rfloor$. Including $\tau_{l,j}$ itself, at most $\left\lfloor \frac{r_{l,j}-t_0}{T_l} \right\rfloor + 1$ jobs of τ_l create demand in the interval. Thus, in the time interval $[t_0, t_c)$, the demand due to τ_l on processor s_p is at most

$$\begin{aligned} \left(\left\lfloor \frac{r_{l,j} - t_0}{T_l} \right\rfloor + 1 \right) C_l &\leq \left(\frac{r_{l,j} - t_0}{T_l} + 1 \right) C_l \\ &= \{\text{by (4.1)}\} \\ & u_l (r_{l,j} - t_0) + C_l. \end{aligned} \tag{4.11}$$

For the purpose of minimizing redundancy in expressions, we define

$$K = \psi_{h,p} \cdot (2T_h + \Delta_h) + 2C_h + C_l. \tag{4.12}$$

Using this definition, by (4.10), (4.11), and (4.12), the total demand within $[t_0, t_c)$ due to migrating tasks is less than

$$
\begin{aligned}
& K + \psi_{h,p} \cdot (t_c - t_0) + u_l (r_{l,j} - t_0) \\
= & \{\text{rearranging}\} \\
& K + \psi_{h,p} \cdot (t_c - r_{l,j}) + (\psi_{h,p} + u_l)(r_{l,j} - t_0) \\
< & \{\text{by Property 4.5}\} \\
& K + \psi_{h,p} \cdot (t_c - r_{l,j}) + s_p (r_{l,j} - t_0).
\end{aligned}
$$

Since for contradiction, we assumed that $\tau_{l,j}$ has lateness exceeding Δ_l, *i.e.*, $\tau_{l,j}$ completes execution after t_c, the total demand in the time interval $[t_0, t_c)$ is greater than the total supply in this interval, which is $s_p(t_c - t_0)$. This implies

$$K + \psi_{h,p} \cdot (t_c - r_{l,j}) + s_p (r_{l,j} - t_0) > s_p (t_c - t_0). \tag{4.13}$$

By simplifying (4.13), we have

$$K > (t_c - r_{l,j})(s_p - \psi_{h,p}). \tag{4.14}$$

By Property 4.5, we have $\psi_{h,p} + u_l < s_p$. Because $u_l > 0$, this implies $\psi_{h,p} < s_p$, *i.e.*,

$$s_p - \psi_{h,p} > 0. \tag{4.15}$$

By (4.14) and (4.15),

$$t_c - r_{l,j} < \frac{K}{s_p - \psi_{h,p}}. \tag{4.16}$$

Replacing the right-hand side of (4.16) by the definition of K in (4.12) and the definition of Δ_l in (4.9), we have

$$t_c - r_{l,j} < \Delta_l + T_l. \tag{4.17}$$

Since $T_l = d_{l,j} - r_{l,j}$, (4.17) implies $t_c < d_{l,j} + \Delta_l$, which contradicts the definition of t_c Thus, such an assumed job $\tau_{l,j}$ with lateness exceeding Δ_l does not exist. Hence, Theorem 4.1 holds. □

4.2.2.2 Fixed Tasks

In this subsection, instead of lateness, we consider tardiness directly.

To begin with, note that if no migrating task assigns jobs to a processor, then all of the fixed tasks on that processor have a tardiness bound of zero, since EDF is optimal for uniprocessor scheduling and by Property 4.2 we do not overutilize any single processor.

Theorem 4.2 below provides a tardiness bound for a fixed task that executes on a processor where migrating task(s) also execute. In this case, the tardiness bound for the fixed task depends on the lateness bound(s) for the migrating task(s) on the same processor, which can be computed by Theorem 4.1. By Property 4.1, at most two migrating tasks have non-zero shares on a processor.

Theorem 4.2. *Suppose that one or two migrating tasks have non-zero shares on processor s_p. If two, let τ_l (τ_h) be the one with lower (higher) priority; if only one, let τ_l denote that task and consider τ_h to be a "null" task with $C_h = 0$, $\psi_{h,p} = 0$, and $T_h = 1$. Then, a fixed task τ_i on s_p has tardiness at most*

$$\Delta_i = \frac{\psi_{l,p} \cdot (2T_l + \Delta_l) + 2C_l + \psi_{h,p} \cdot (2T_h + \Delta_h) + 2C_h}{s_p - \psi_{l,p} - \psi_{h,p}}. \tag{4.18}$$

Proof. This proof is similar to that of Theorem 4.1.

Interval $[t_0, t_c)$. Let $\tau_{i,j}$ be the first job of any fixed task on s_p that has tardiness exceeding the bound in (4.18) and define $t_c \stackrel{\text{def}}{=} d_{i,j} + \Delta_i$. Let t_0 be the latest idle time instant before t_c, *i.e.*, all jobs that were released before t_0 and with a priority at least $\tau_{i,j}$'s priority have completed execution by t_0 and at least one job with a priority at least $\tau_{i,j}$'s priority is released at t_0. t_0 is well-defined because if no such time instant exists within $(0, t_c)$, then time 0 must be a such time instant.

Demand from mirgrating tasks. By Lemma 4.1, in $[t_0, t_c)$, the demand from τ_l on s_p is less than

$$\psi_{l,p} \cdot (t_c - t_0) + \psi_{l,p} \cdot (2T_l + \Delta_l) + 2C_l, \tag{4.19}$$

and the demand from τ_h on s_p is less than

$$\psi_{h,p} \cdot (t_c - t_0) + \psi_{h,p} \cdot (2T_h + \Delta_h) + 2C_h. \tag{4.20}$$

Demand from fixed tasks. A fixed task τ_k can release at most $\left\lfloor \frac{d_{i,j}-t_0}{T_k} \right\rfloor$ jobs with a priority at least $\tau_{i,j}$'s priority in the interval $[t_0, t_c)$. Thus, the demand from fixed tasks in $[t_0, t_c)$ is at most

$$\begin{aligned}\sum_{\tau_k \in \tau_p^f} \left\lfloor \frac{d_{i,j}-t_0}{T_k} \right\rfloor \cdot C_k \leq & (d_{i,j}-t_0) \cdot \sum_{\tau_k \in \tau_p^f} \frac{C_k}{T_k} \\ = & \{\text{by the definition of } \sigma_p^f\} \\ & (d_{i,j}-t_0) \cdot \sigma_p^f \\ \leq & \{\text{by Property 4.2}\} \\ & (d_{i,j}-t_0)(s_p - \psi_{l,p} - \psi_{h,p}). \end{aligned} \tag{4.21}$$

For the purpose of minimizing redundancy in expressions, we define

$$K = \psi_{l,p} \cdot (2T_l + \Delta_l) + 2C_l + \psi_{h,p} \cdot (2T_h + \Delta_h) + 2C_h. \tag{4.22}$$

Using this definition, by (4.19), (4.20), (4.21), and (4.22), the total demand within $[t_0, t_c)$ is at most

$$K + (\psi_{l,p} + \psi_{h,p})(t_c - t_0) + (s_p - \psi_{l,p} - \psi_{h,p})(d_{i,j} - t_0).$$

Since for the purpose of contradiction, we assume $\tau_{i,j}$ has tardiness exceeding Δ_i, *i.e.*, $\tau_{i,j}$ completes execution after t_c, the total demand in the time interval $[t_0, t_c)$ is greater than the total supply in the interval which is $s_p(t_c - t_0)$. That is,

$$K + (\psi_{l,p} + \psi_{h,p})(t_c - t_0) + (s_p - \psi_{l,p} - \psi_{h,p})(d_{i,j} - t_0) > s_p(t_c - t_0). \tag{4.23}$$

By simplifying (4.23), we have

$$K > (t_c - d_{i,j})(s_p - \psi_{l,p} - \psi_{h,p}). \tag{4.24}$$

By Property 4.2, we have $\sigma_p^f + \psi_{h,p} + \psi_{l,p} \leq s_p$. Because $\sigma_p^f > 0$, this implies

$$s_p - \psi_{l,p} - \psi_{h,p} \geq \sigma_p^f > 0. \tag{4.25}$$

By (4.24) and (4.25),

$$t_c - d_{i,j} < \frac{K}{s_p - \psi_{l,p} - \psi_{h,p}}. \tag{4.26}$$

Replacing the right-hand side of (4.26) by the definition of K in (4.22) and the definition of Δ_i in (4.18), we have

$$t_c - d_{i,j} < \Delta_i. \tag{4.27}$$

(4.27) implies $t_c < d_{i,j} + \Delta_i$, which contradicts the definition of t_c. Thus, such an assumed job $\tau_{i,j}$ with tardiness exceeding Δ_i does not exist. Hence, Theorem 4.2 holds. □

4.2.3 Evaluation

To evaluate how restrictive the assumed per-task utilization constraint is and the effectiveness of EDF-sh, we conducted experiments to assess schedulability and tardiness bounds for EDF-sh.

When conducting such experiments for identical multiprocessors, the assumed platform is implicitly determined by an assumed total processor capacity, or the number of processors. However, for uniform multiprocessors, processor speeds must be defined. Given a total processor capacity, there are a infinite number of speed choices from which to select. Because only selected choices can be considered, no evaluation can be exhaustive. In our experiments, we considered systems of eight processors with a total processor capacity of 36. We considered four such platforms, with speeds as follows: $\pi_1 = \{6,6,6,6,3,3,3,3\}$, $\pi_2 = \{8,8,4,4,4,4,2,2\}$, $\pi_3 = \{8,7,6,5,4,3,2,1\}$, $\pi_4 = \{15,3,3,3,3,3,3,3\}$.

The process of randomly generating feasible task systems for the considered platforms also varies from that for identical ones. For feasibly scheduling tasks on an identical multiprocessor, the per-task utilization bound is fixed for every task to be 1.0. However, per-task utilization bounds for feasibly scheduling tasks on a uniform multiprocessor must instead follow (4.3). As such, before generating a new task, we calculated a per-task utilization cap for it by (4.3), considering previously generated tasks. We then selected the utilization of that task uniformly at random between zero and the computed cap. This generation process terminates when the total utilization of all generated tasks exceeds or equals a pre-set total utilization limit. The utilization of the last generated task is then adjusted so that the total generated utilization matches the pre-set limit.

We require the number of tasks n to be at least the number of processors m. To ensure this, whenever $n < m$ held for a generated system, a task was chosen at random and replaced by two tasks with half the

utilization of the original one (this process was repeated as necessary). Given a platform and a total task system utilization, having fewer (more) tasks means having higher (lower) expected per-task utilizations. To reflect these two extremes, we defined the minimum number of the tasks to be either eight (fewer but heavier tasks) or 32 (more but lighter tasks) for every considered platform. Also, we selected each task's execution requirement uniformly from $[5, 25]$ and calculated its period from its utilization and execution requirement. In all experiments in this section, we varied total utilization within $[0, 36]$ by increments of 0.5, and for each total utilization, we generated $10,000$ feasible task sets.

We compare the results with EDF-ms (Leontyev and Anderson, 2007a), which also supports processors of different speeds as reviewed in Section 2.3.3.

Schedulability.

EDF-sh has the same utilization restrictions (*i.e.*, (4.2) and (4.4)) as EDF-ms, but EDF-sh can support platforms in which speed groups exist with only one processor, while EDF-ms requires each such group to have at least two processors. For this reason, EDF-sh dominates EDF-ms in terms of SRT schedulability.

Given this provable dominance over EDF-ms, our assessment of schedulability under EDF-sh focuses on determining the fraction of randomly generated feasible systems (as defined by (4.2) and (4.3)) it can successfully schedule for every given total utilization and every platform. Figure 4.3 shows the results of these experiments. More than 87% of the generated systems were SRT-schedulable under EDF-sh. In general, the smaller the difference among processor speeds, the better the schedulability. This makes sense, since for identical multiprocessors, EDF-sh is SRT-optimal, like EDF-os. Furthermore, when there are many lighter tasks instead of few heavier ones, schedulability is quite close to optimal.

Tardiness Bounds.

We also compared tardiness bounds under EDF-sh to those under EDF-ms. Since EDF-ms requires each speed group to have at least two processors, *i.e.*, it does not apply to π_3 and π_4, we only computed tardiness bounds for π_1 and π_2. We compared EDF-sh and EDF-ms in terms of both maximum absolute tardiness bounds and maximum relative tardiness bounds, where the latter is defined as the ratio of a task's tardiness bound to its period. Figure 4.4 shows absolute tardiness bounds and Figure 4.5 shows relative tardiness bounds.

In most cases, EDF-sh exhibits significantly lower maximum tardiness bounds than EDF-ms. The only exception to this is when total utilization is close to overutilizing the platform, and even then, EDF-sh is never substantially worse.

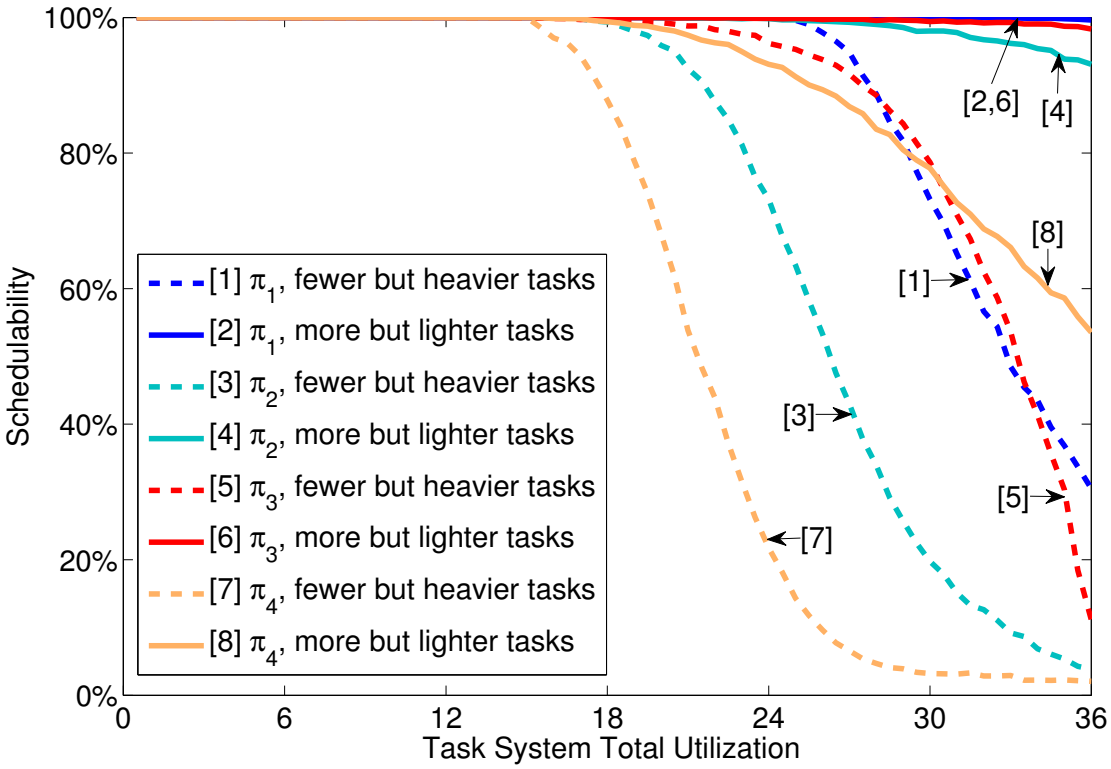


Figure 4.3: Schedulability under EDF-sh.

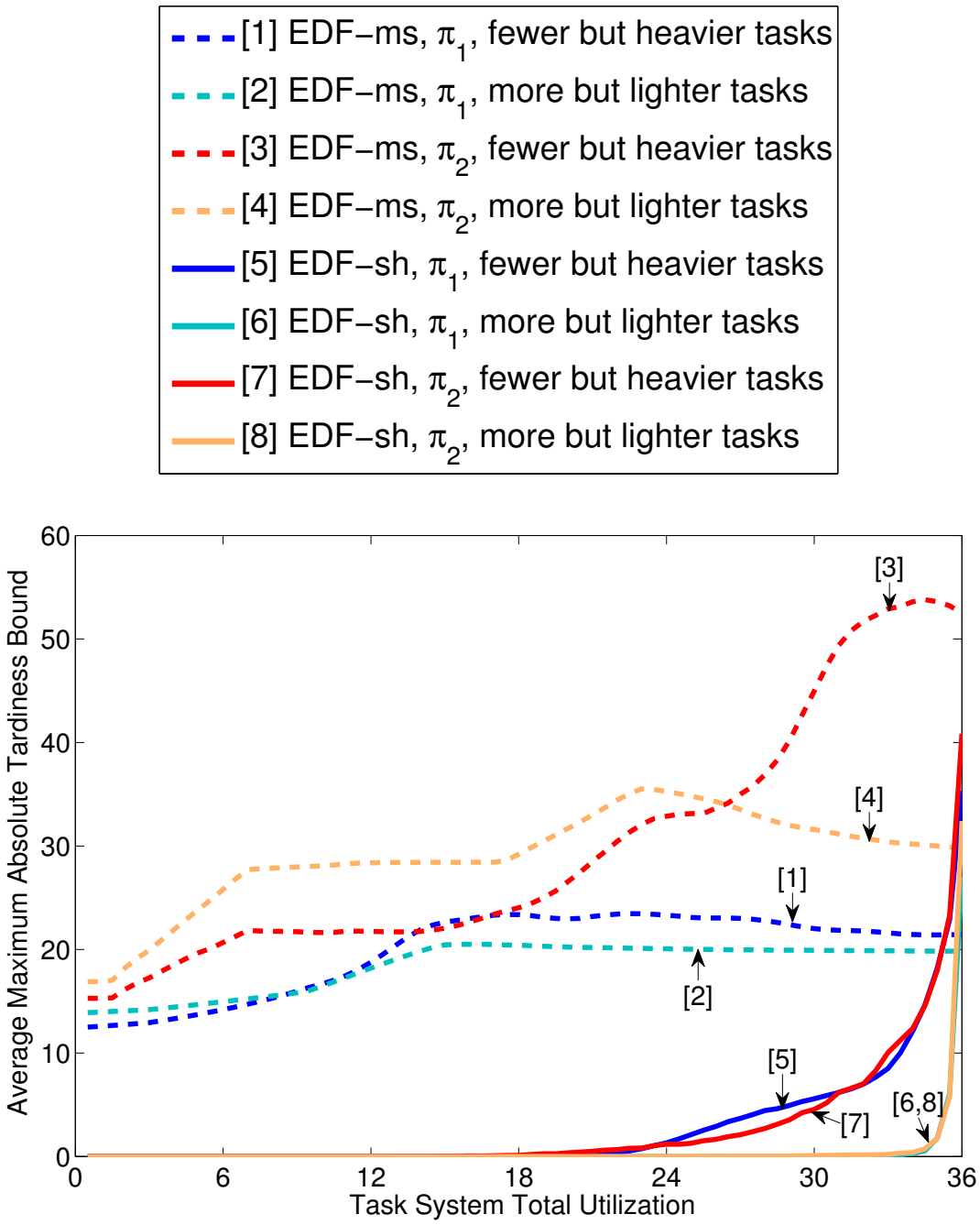


Figure 4.4: Absolute tardiness bounds of EDF-ms and EDF-sh.

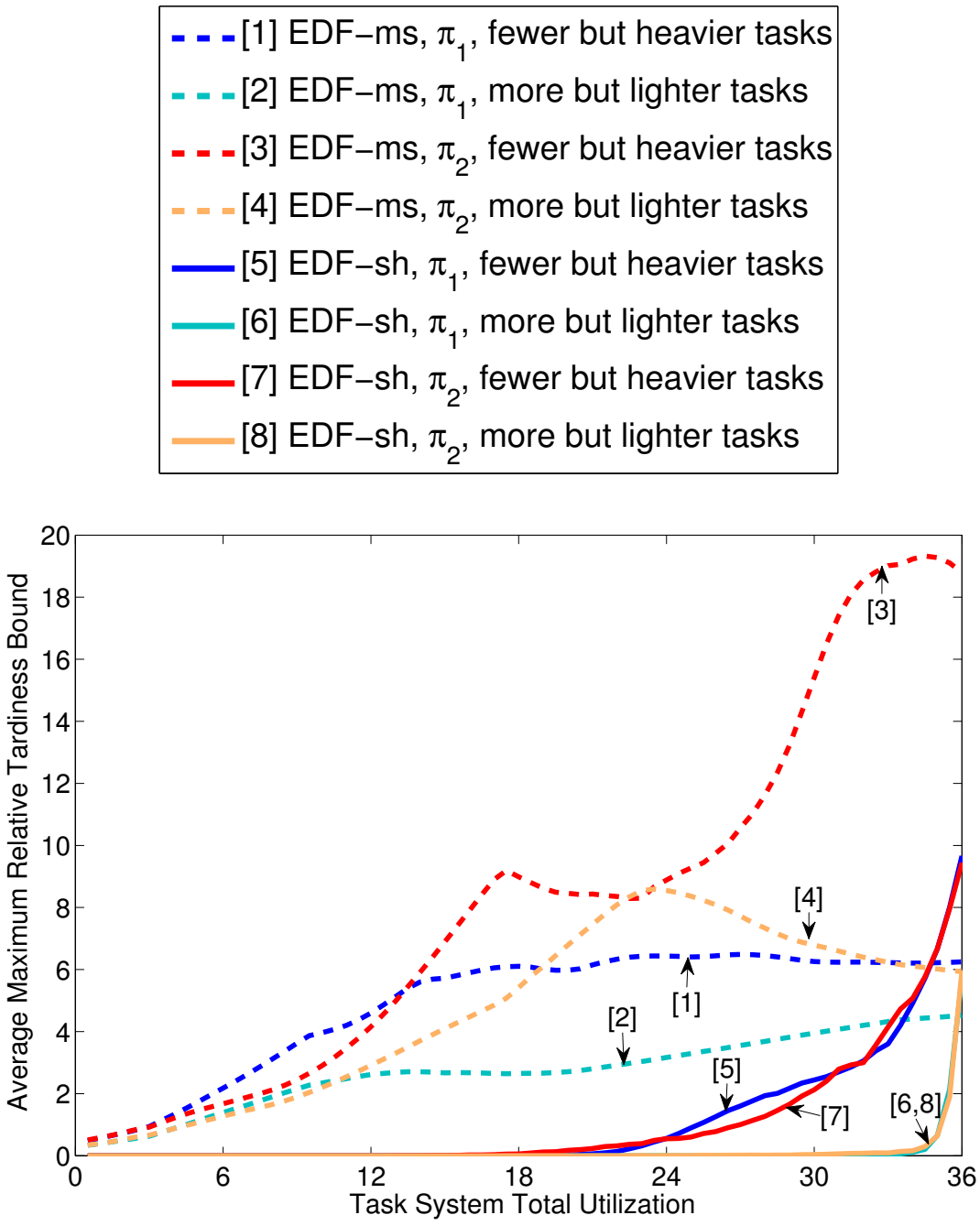


Figure 4.5: Relative tardiness bounds of EDF-ms and EDF-sh.

4.3 EDF-tu

In this section, we present the second algorithm, EDF-tu. In contrast to EDF-sh, EDF-tu is SRT-optimal and maybe HRT-optimal under certain settings. We therefore discuss feasible task assignments, which are an important concept towards these optimality results (Section 4.3.1). Next, we present algorithm EDF-tu by describing its assignment phase and execution phase (Section 4.3.2). Then, we prove both the HRT- and SRT-optimality of EDF-tu (Section 4.3.3), show the tightness of our task assignment scheme (Section 4.3.4), and present an evaluation (Section 4.3.5).

4.3.1 Feasible Assignments

Most semi-partitioned scheduling algorithms are defined by specifying separate *assignment* and *execution* phases. In the former, per-processor shares are defined offline for each task, and fixed tasks are distinguished from migrating ones. In the latter, an actual schedule is produced at runtime, based on the task share assignments. In this section, we explore the problem of obtaining share assignments. We show that issues arise in the case of uniform platforms that have not been considered before.

In addressing such issues, we will need to examine situations where some number of tasks in the assignment process have been assigned as fixed. We let σ_i^f denote the sum of the utilizations of the fixed tasks on processor s_i, *i.e.*,

$$\sigma_i^f = \sum_{\substack{\tau_k \text{ is a fixed task} \\ \text{on processor } s_i}} u_k. \tag{4.28}$$

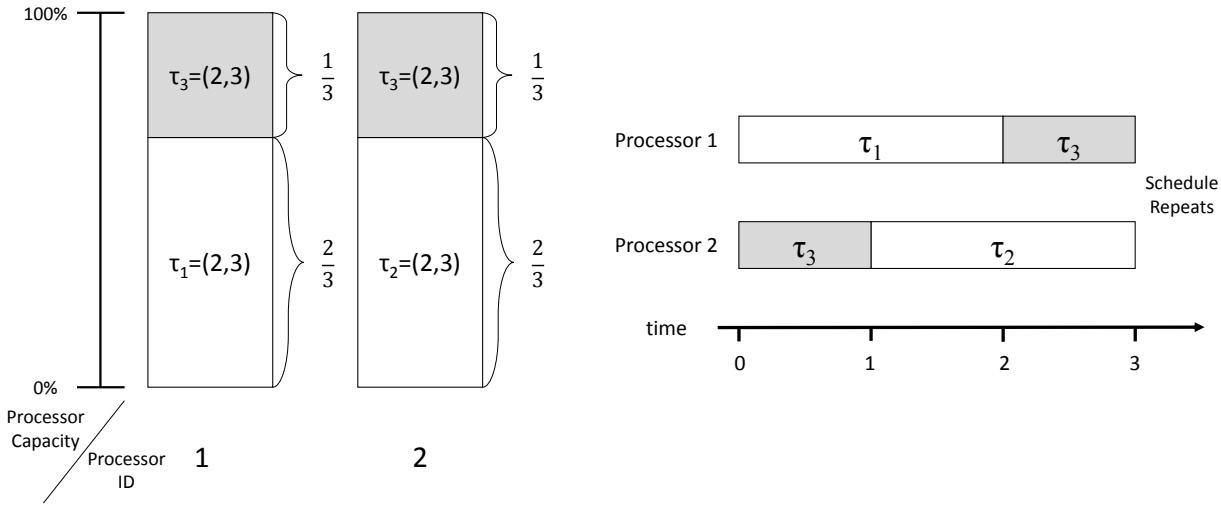
We define the *residual capacity* (*i.e.*, the currently available capacity) of processor s_i as $s_i - \sigma_i^f$.

In most prior work on semi-partitioned scheduling on *identical* platforms, a greedy assignment method is used wherein the currently considered task is assigned as fixed if possible. Consider the following example.

Example 4.3. Three tasks $\{\tau_1 = (2,3),\ \tau_2 = (2,3),\ \tau_3 = (2,3)\}$ are to be scheduled on two unit-speed processors. We greedily assign the first two tasks as fixed, and then require the remaining one to migrate. This results in the share assignment shown in Figure 4.6 (a). As seen in Figure 4.6 (b), we can easily determine a schedule corresponding to this assignment such that all deadlines are met. ◊

As the next example shows, a greedy assignment strategy can be problematic on a uniform platform.

Example 4.4. Consider scheduling the two tasks $\{\tau_1 = (2,1),\ \tau_2 = (2,1)\}$ on the two-processor uniform platform $\pi = \{s_1 = 3,\ s_2 = 1\}$. When we first consider assigning τ_1, processor s_1 has enough capacity for it,



(a) assignment (b) schedule

Figure 4.6: Assignment and schedule for Example 4.3.

and if we assign τ_1 there, the residual capacity of the system matches the utilization of τ_2. The task share allocations must be as shown in Figure 4.7 (a). The allocation to τ_2 implies that it must execute in parallel as shown in Figure 4.7 (b), so this assignment is infeasible. However, the original system is feasible, as seen in insets (c) and (d) of Figure 4.7. ◊

We now determine conditions for ensuring that a task assignment is feasible.

After some tasks have been fixed, let $\{z_i\}$ denote the residual capacities of the processors on platform π, indexed in non-increasing order. Note that the indexing of $\{z_i\}$ may differ from that of $\{s_i\}$, *i.e.*, z_i does not necessarily correspond to the residual capacity on s_i. Let $p(i)$ denote the index of the processor with the remaining capacity z_i, *i.e.*, z_i is the remaining capacity of the processor of speed $s_{p(i)}$: $z_i + \sigma^f_{p(i)} = s_{p(i)}$. Also, let $Z_k = \sum_{i=1}^{k} z_i$.

Theorem 4.3. *Any task set that is feasible on the fully available platform $\pi' = \{s'_1 = z_1, s'_2 = z_2, \ldots, s'_m = z_m\}$ can also be correctly scheduled using the residual capacities $\{z_i\}$ of platform π. (In a correct schedule, all deadlines are met and all requirements of the sporadic model are respected.)*

Proof. We prove this theorem by transforming an arbitrary schedule $\mathcal{S}'$ on π' to a corresponding schedule $\mathcal{S}$ on π such that if $\mathcal{S}'$ is correct, then $\mathcal{S}$ is also correct. Moreover, in $\mathcal{S}$, only a capacity of z_i is utilized on $s_{p(i)}$ for each i.

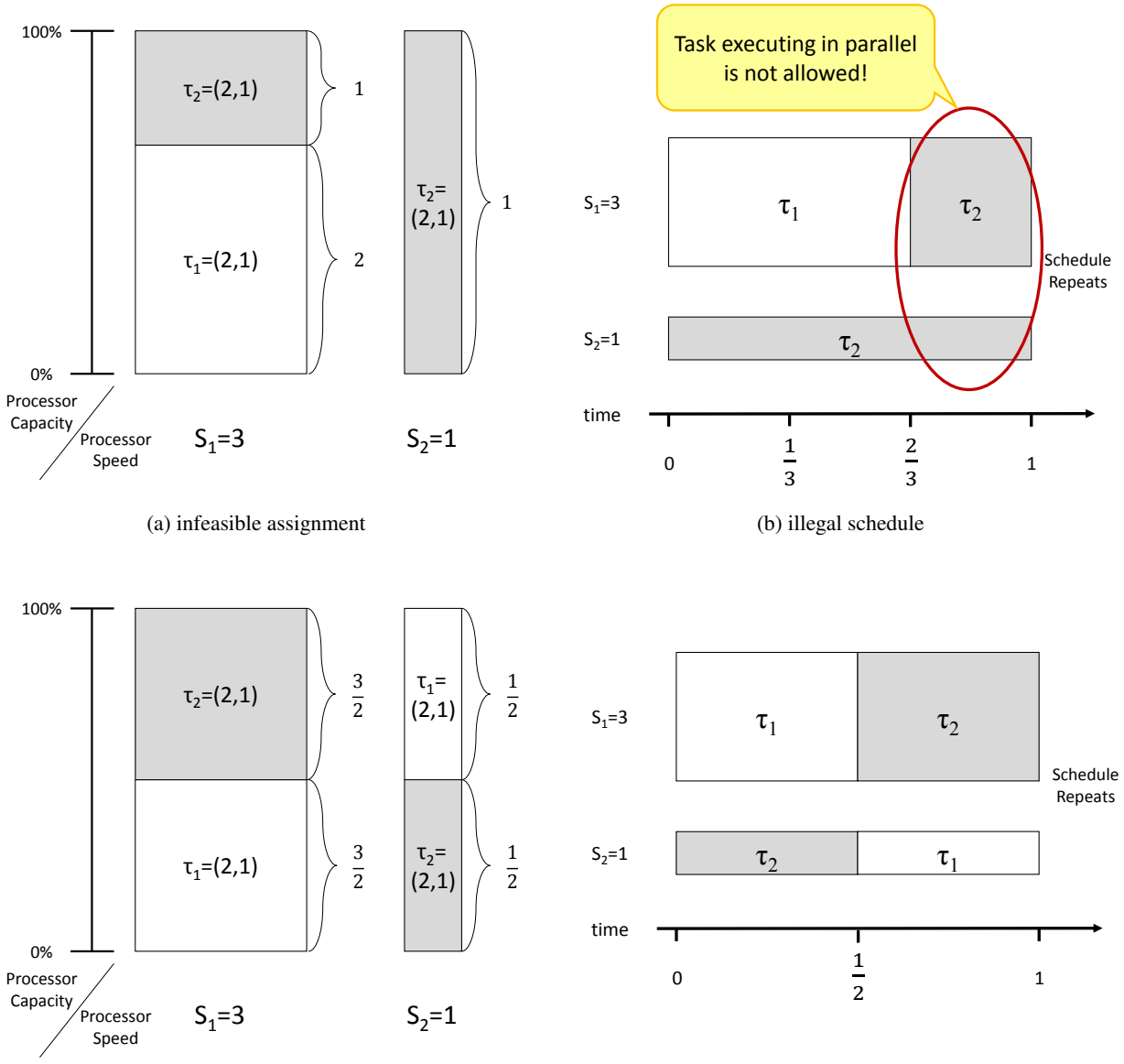


Figure 4.7: Assignment and schedule for Example 4.4. The width of each rectangle represents the speed of its corresponding processor.

We split the time line of $\mathcal{S}'$ into slices of width Δ such that, on *any* processor of π', *all* preemptions, migrations, job releases, and job deadlines occur on slice boundaries. This requirement can be met by choosing Δ small enough. We construct $\mathcal{S}$ on a per-slice basis: within each slice, we schedule in $\mathcal{S}$ exactly the same jobs as scheduled in $\mathcal{S}'$ and on exactly the same processors. However, in $\mathcal{S}$, the job (if any) that executes on processor s'_i in $\mathcal{S}'$ is scheduled within the first $\frac{z_i}{s_{p(i)}} \cdot \Delta$ time units of the slice. It is easy to see that the resulting schedule $\mathcal{S}$ is correct if $\mathcal{S}'$ is. In particular, all deadlines will be met and all requirements of the sporadic model are respected (including no intra-task parallelism). Also, it is straightforward to see that only a capacity of z_i is utilized on $s_{p(i)}$ for each i. □

By Theorem 4.3, after some tasks have been assigned as fixed, the platform defined by the resulting residual capacities can be viewed as a fully available platform as far as the feasibility of the remaining, unassigned tasks is concerned.

In the rest of this section, let τ denote the set of the remaining, unassigned tasks, and let π denote the platform defined by the residual processor capacities. Also, let $\{u_i\}$ denote the utilizations of the remaining tasks, and assume they are indexed in non-increasing order. Let $U_k = \sum_{i=1}^{k} u_i$. Define the total utilization of the remaining tasks as U_τ and the total residual capacity of the platform as Z_π.

Theorem 4.4. *τ is feasible on π if the following conditions hold.*

$$U_\tau \leq Z_\pi \tag{4.29}$$

$$U_k \leq Z_k \quad \text{for} \quad k = 1, 2, \cdots, m-1 \tag{4.30}$$

Proof. Follows from Theorem 4.3 and the feasibility condition for uniform platforms, (4.2) and (4.3). □

Note that just "if" is stated in Theorem 4.4 and "only if" cannot be asserted, *i.e.*, (4.29) and (4.30) are only a sufficient condition for feasibility. This is because (4.30) is not necessary for feasibility, although the similar condition (4.3) is. We show this by the following counterexample.

Example 4.5. Consider scheduling a single task $\tau_1 = (2, 1)$ on two unit-speed processors. This system is clearly infeasible since $u_1 = 2 > 1$ holds, which violates (4.3).

Now, consider scheduling the same task τ_1 on two other processors, where both processors have a residual capacity of 1, *i.e.*, $z_1 = z_2 = 1$. Then, this system violates (4.30), but it could be feasible. For example,

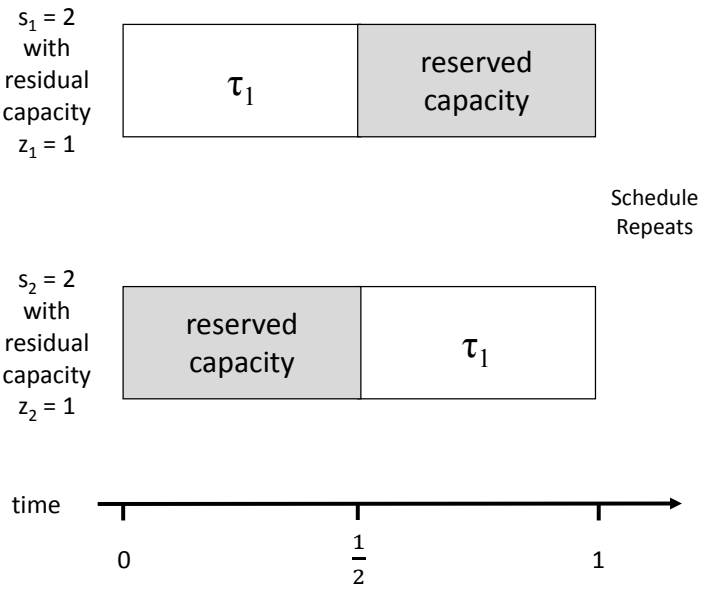


Figure 4.8: A correct schedule for Example 4.5.

assuming the two processors both have an initial speed of 2, Figure 4.8 is a correct schedule for this system. ◊

Nonetheless, by Theorem 4.4, we have a sufficient test to check if a task assignment is guaranteed to preserve the feasibility of the system. Next, we show that the following scheme can guarantee that any feasible system can have at most m migrating tasks and still be feasible.

- Consider tasks from lightest to heaviest by utilization.
- Use the best-fit bin-packing heuristic to assign as many tasks as possible as fixed.

The guarantee mentioned above follows from the following lemma. (Note that the task τ_n mentioned in the lemma definitely can be assigned as fixed if (4.29) and (4.30) hold.)

Lemma 4.4. *Let n denote the number of tasks in τ and assume $n \geq m+1$. If τ and the residual processor capacities π satisfy* (4.29) *and* (4.30)*, then after using the best-fit heuristic to assign the lightest task in τ (i.e., τ_n) as fixed, the remaining task set τ' and residual processor capacities π' must satisfy* (4.29) *and* (4.30) *as well.*

Proof. We prove this lemma by contradiction by showing that any violation of (4.29) or (4.30) for τ' and π' implies a violation of (4.29) or (4.30) for τ and π.

Since we consider tasks from lightest to heaviest, the order of tasks in τ' is the same as that in τ except the absence of the lightest one, τ_n. Thus,

$$u'_i = u_i \quad \text{for } 1 \leq i \leq n-1. \tag{4.31}$$

Case 1: τ' and π' violate (4.29). Since the only change is that τ_n is assigned, the total remaining utilization decreases by u_n and total residual capacity decreases by u_n too, *i.e.*, $U'_{\tau'} = U_\tau - u_n$ and $Z'_{\pi'} = Z_\pi - u_n$. Thus, τ' and π' violating (4.29) implies τ and π violate (4.29) as well.

Case 2: τ' and π' violate (4.30). In this case, let z_γ be the processor on which τ_n is fixed. Since $\{z_i\}$ is indexed non-increasingly, without loss of generality, we can assume that the best-fit bin-packing heuristic always chooses the highest-indexed z_i among those with equal values (if it does not, we can re-index them, which will not change either $\{z_i\}$ or the resulting $\{z'_i\}$), *i.e.*, $z_\gamma > z_{\gamma+1}$ if $\gamma < m$. Thus, as a result of the best-fit heuristic, we have

$$u_n > z_i \quad \text{for any } i > \gamma. \tag{4.32}$$

Moreover, the assignment of τ_n will not alter the indices of the largest $\gamma - 1$ capacities in π, *i.e.*,

$$z'_i = z_i \quad \text{for any } i \leq \gamma - 1. \tag{4.33}$$

Also, other than z_γ, the relative ordering in $\{z_i\}$ is preserved in $\{z'_i\}$ as well. That is, letting ϕ denote the new index of z_γ in π', *i.e.*, $z'_\phi = z_\gamma - u_n$, where $\gamma \leq \phi \leq m$, $\{z'_i\}$ is

$$\{z_1, \ldots, z_{\gamma-1}, z_{\gamma+1}, \ldots, z_\phi, z_\gamma - u_n, z_{\phi+1}, \ldots, z_m\}. \tag{4.34}$$

Note that, if $\phi = \gamma$, then the sequence $z_{\gamma+1}$, …, z_ϕ is empty; similarly, $\phi = m$ implies that the sequence $z_{\phi+1}$, …, z_m is empty.

From (4.34),

$$z'_i = \begin{cases} z_i & \text{if } 1 \leq i \leq \gamma - 1 \text{ or } i \geq \phi + 1, \\ z_{i+1} & \text{if } \gamma \leq i \leq \phi - 1, \\ z_\gamma - u_n & \text{if } i = \phi. \end{cases} \tag{4.35}$$

Case 2.1: τ' and π' violate (4.30) at k such that $k \leq \gamma - 1$. By (4.31) and (4.35), $U'_k = U_k$ and $Z'_k = Z_k$. Thus, τ and π violate (4.30) at k as well.

Case 2.2: τ' and π' violate (4.30) at k such that $k \geq \gamma$.That is,

$$U'_k > Z'_k. \tag{4.36}$$

First, we show the following inequality holds in Case 2.2 by considering two sub-cases.

$$Z'_k \geq \left(\sum_{i=1}^{k} z_i\right) - u_n. \tag{4.37}$$

Case 2.2.1: $\gamma \leq k \leq \phi - 1$.

$$\begin{aligned}
Z'_k &= \left(\sum_{i=1}^{\gamma-1} z'_i\right) + \left(\sum_{i=\gamma}^{k-1} z'_i\right) + z'_k \\
&\geq \{\text{since } k \leq \phi - 1 \text{ and } \{z'_i\} \text{ is in non-increasing order}\} \\
&\quad \left(\sum_{i=1}^{\gamma-1} z'_i\right) + \left(\sum_{i=\gamma}^{k-1} z'_i\right) + z'_\phi \\
&= \{\text{by (4.35)}\} \\
&\quad \left(\sum_{i=1}^{\gamma-1} z_i\right) + \left(\sum_{i=\gamma}^{k-1} z_{i+1}\right) + (z_\gamma - u_n) \\
&= \{\text{simplifying}\} \\
&\quad \left(\sum_{i=1}^{k} z_i\right) - u_n.
\end{aligned}$$

Case 2.2.2: $k \geq \phi$.

$$\begin{aligned}
Z'_k &= \left(\sum_{i=1}^{\gamma-1} z'_i\right) + \left(\sum_{i=\gamma}^{\phi-1} z'_i\right) + z'_\phi + \left(\sum_{i=\phi+1}^{k} z'_i\right) \\
&= \{\text{by (4.35)}\} \\
&\quad \left(\sum_{i=1}^{\gamma-1} z_i\right) + \left(\sum_{i=\gamma}^{\phi-1} z_{i+1}\right) + (z_\gamma - u_n) + \left(\sum_{i=\phi+1}^{k} z_i\right) \\
&= \{\text{simplifying}\}
\end{aligned}$$

$$\left(\sum_{i=1}^{k} z_i\right) - u_n.$$

From these sub-cases, we can conclude that (4.37) holds. By (4.36) and (4.37), we have

$$U'_k + u_n > \left(\sum_{i=1}^{k} z_i\right). \tag{4.38}$$

By the condition of Case 2.2, $k \geq \gamma$, and by (4.32), we have $u_n > z_i$ for any $i \geq k+1 > \gamma$, which implies

$$(m-k)u_n > \left(\sum_{i=k+1}^{m} z_i\right). \tag{4.39}$$

By (4.38), (4.39), and the definition of Z_π,

$$U'_k + (m-k+1)u_n > Z_\pi. \tag{4.40}$$

Finally, we have

$$\begin{aligned}
U_\tau = \quad & U_k + \sum_{i=k+1}^{n} u_i \\
= \quad & \{\text{by (4.31)}\} \\
& U'_k + \sum_{i=k+1}^{n} u_i \\
\geq \quad & \{\text{since } \{u_i\} \text{ is in non-increasing order}\} \\
& U'_k + (n-k)u_n \\
\geq \quad & \{\text{since } n \geq m+1\} \\
& U'_k + (m-k+1)u_n.
\end{aligned} \tag{4.41}$$

By (4.40) and (4.41), $U_\tau > Z_\pi$ holds, *i.e.*, τ and π violate (4.29). □

In the remainder of this section for EDF-tu, we say that an assignment of a task as fixed to a processor is *legal* if and only if (4.29) and (4.30) hold for the remaining, unassigned tasks.

Theorem 4.5. *For any feasible task system, if we continue to assign tasks as fixed as long as legal assignments can be made using the best-fit heuristic, with tasks considered from lightest to heaviest by utilization, then at most m tasks will remain as unassigned.*

Proof. By Theorem 4.4 and Lemma 4.4, we can continue to make legal assignments at least until the number of unassigned tasks is m. □

4.3.2 Algorithm EDF-tu

We now describe our new scheduling algorithm EDF-tu by considering its assignment and execution phases separately.

Assignment phase. The assignment phase must not only distinguish fixed tasks from migrating ones, but also determine the per-processor share allocations for each migrating task. As for determining which tasks should be fixed, Theorem 4.5 suggests the way forward: we simply consider tasks from lightest to heaviest by utilization, and keep assigning tasks as fixed via the best-fit heuristic until all of them are assigned or we encounter a task that cannot be so assigned legally. The remaining m' unassigned tasks will be migrating tasks. By Theorem 4.5, $m' \leq m$. Also, by Theorems 4.3 and 4.5, the set of migrating tasks is feasible on the resulting platform as defined by the residual processor capacities. In fact, this set of tasks is feasible on the sub-platform comprised of the m' processors with the largest residual capacities.

In order to determine per-processor share allocations for migrating tasks, and how such tasks are scheduled alongside fixed ones, we construct a *processor allocation table*. This table indicates which task may execute on which processor within an interval of time, or *frame*, of length F. As shown later in Section 4.3.3, if HRT-schedulability is the goal, then the frame size F must meet a certain constraint, but this constraint is not required if only SRT-schedulability is required.

We construct the processor allocation table A via a two-step process (which is illustrated via an example below). In the first step, we construct a processor allocation table A' for the m' migrating tasks on a hypothetical platform $\pi' = \{s'_1 = z_1, s'_2 = z_2, \ldots, s'_{m'} = z_{m'}\}$ by applying the Level Algorithm to schedule the job set $\mathcal{J}$ with execution costs $\{u_1 \cdot F, u_2 \cdot F, \ldots, u_{m'} \cdot F\}$ on π'. We obtain the table A' by allocating processor s'_i to task τ_k in each sub-interval where the corresponding job of cost $u_k \cdot F$ executes on processor s'_i. The Level Algorithm ensures that the schedule for $\mathcal{J}$ is free of intra-job parallelism. This implies that task allocations in the table A' are free of intra-task parallelism. Also, by Theorems 2.1, 4.4, and 4.5, the

makespan of the schedule for $\mathcal{J}$ is at most F. This implies that A' gives task allocations over an interval of length at most F as well. The total allocation recorded for each migrating task τ_k in A' is $u_k \cdot F$.

In the second step, we obtain the final table A by integrating allocations for fixed tasks into A'. Examining the task allocations recorded in A', we say that the sub-interval $[t_1, t_2)$ is a *maximal non-preemptive sub-interval on processor* s'_i if s'_i is allocated to the same migrating task throughout $[t_1, t_2)$ and s'_i is not allocated to that task either immediately before t_1 or at t_2. We construct the processor allocation table A for the real physical platform π from A' by examining all such maximal non-preemptive sub-intervals. In particular, if the migrating task τ_k is allocated in A' to processor s'_i throughout the maximal non-preemptive sub-interval $[t_1, t_2)$, then we allocate processor $s_{p(i)}$ to τ_k in A throughout the first $\frac{z_i}{s_{p(i)}}$ of the maximal non-preemptive sub-interval, *i.e.*, $\left[t_1, t_1 + \frac{z_i}{s_{p(i)}} \cdot (t_2 - t_1)\right)$. We allocate the remainder of the maximal non-preemptive sub-interval, *i.e.*, $\left[t_1 + \frac{z_i}{s_{p(i)}} \cdot (t_2 - t_1), t_2\right)$, to fixed tasks on $s_{p(i)}$. We denote this in the table by indicating that the sub-interval $\left[t_1 + \frac{z_i}{s_{p(i)}} \cdot (t_2 - t_1), t_2\right)$ is allocated to $\sigma^f_{p(i)}$. If $m' < m$, then A is extended to incorporate all processors by fully allocating the processors with residual capacities $z_{m'+1}, \ldots, z_m$ to the fixed tasks assigned to those processors.

The pseudo-code for the assignment process is given in Algorithm 2. The following example provides an illustration.

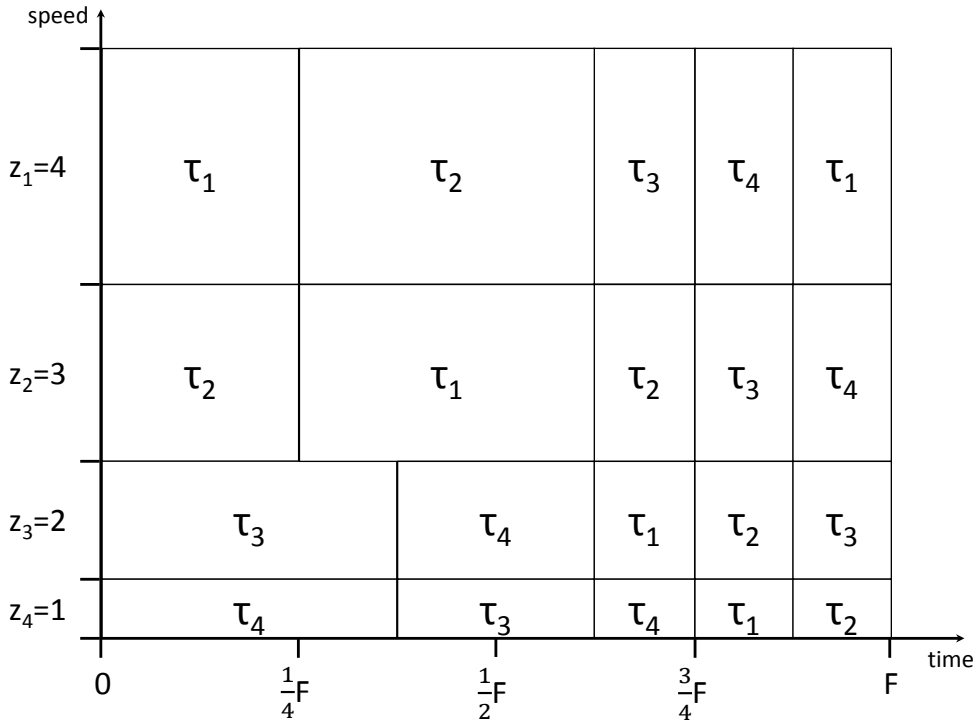
Example 4.6. Suppose that after all fixed tasks have been identified, we are left with four migrating tasks with utilizations {3, 3, 2.125, 1.875} to be scheduled on four processors with residual capacities {4, 3, 2, 1}. Further, suppose we are using a frame size of $F = 4$. Then the schedule resulting from the Level Algorithm within a frame is identical to Example 2.1. Figure 4.9 (a) shows the resulting table A', which provides allocations only for migrating tasks. To obtain the final table A for these four processors, we must integrate fixed tasks. To illustrate this, suppose that the processor with residual capacity $z_4 = 1$ corresponds to a physical processor with speed $s_{p(4)} = 2$, *i.e.*, half of this processor's capacity is reserved for fixed tasks. Then the allocations on this processor within each frame will be as depicted in Figure 4.9 (b). ◊

Execution phase. In the execution phase, the processor allocation table A is consulted on a frame-by-frame basis at runtime to determine which task may execute on which processor at any given time. In particular, the following rules are applied at any time $t \in [k \cdot F, (k+1) \cdot F)$, where $k \in \mathbb{Z}^+$.

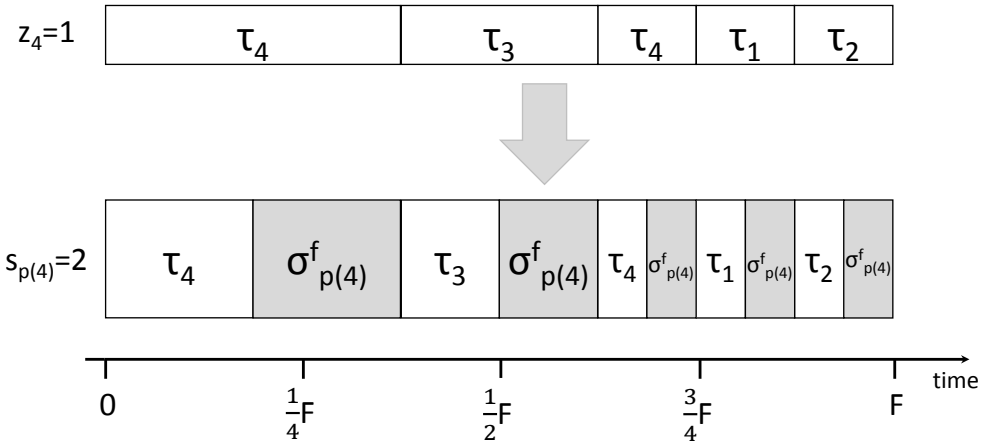
- If processor s'_i is allocated to the migrating task τ_k at time t **mod** F in A, and if τ_k has an unfinished job at time t, then the earliest-released such job is scheduled on processor s'_i.

Algorithm 2 EDF-tu assignment phase

initially $\sigma_p^f := 0$ for all p;
index tasks in non-increasing-utilization order;
index processors in non-increasing-speed order;
/* The first $(n - m)$ lightest tasks are guaranteed to be fixed via the best-fit heuristic. If $n = m$, then this for loop is skipped. */
for $i := n$ **downto** $m+1$ **do**
    Select k that $s_k - \sigma_k^f$ is minimal while at least u_i;
    $\sigma_k^f := \sigma_k^f + u_i$;
end
/* Try to continue fixing tasks until the fixing step is not legal or all tasks are fixed. */
$m' := m$;
$isLegal := 1$;
repeat
    Select k that $s_k - \sigma_k^f$ is minimal while at least $u_{m'}$;
    $last\_\sigma_k^f := \sigma_k^f$;
    $\sigma_k^f = \sigma_k^f + u_{m'}$;
    for $j := 1$ **to** m' **do**
        if $\sum_{j \text{ largest}}(s_p - \sigma_p^f) < \sum_{j \text{ largest}} u_i$ **then**
            $isLegal := 0$;
        end
    end
    $last\_m' := m'$;
    $m' := m' - 1$;
until $isLegal = 0$ ***or*** $m' = 0$;
if $isLegal = 0$ **then**
    /* If the last fixing step is not legal, restore to last feasible assignment. */
    $m' := last\_m'$;
    $\sigma_k^f := last\_\sigma_k^f$;
    /* Now, we have m' migrating tasks to be scheduled on m' processors using a frame-based schedule. */
    for $j := 1$ **to**m' **do**
        $z_j :=$ the j^{th} largest $(s_p - \sigma_p^f)$;
    end
    Use the Level Algorithm to construct the processor allocation table for a frame;
else
    In this case, there is no migrating task, and a valid partitioned schedule can be generated by applying the uniprocessor EDF scheduler on each processor;
end



(a) example for deriving A′



(b) example on one processor for constructing A from A′

Figure 4.9: EDF-tu execution phase illustration for Example 4.6.

- If no such job exists, or if processor s'_i is either unallocated or allocated to σ_i^f at time t **mod** F in A, then an unfinished job of a fixed task on s'_i is scheduled on processor s'_i at time t if one exists. If multiple such jobs exist, then the one with the earliest deadline is selected. If no such job exists, then processor s'_i is idled.

According to the sporadic task model, it is possible for a task to release a job within a frame, *i.e.*, at some time $k \cdot F + r$, where $k \in \mathbb{Z}^+$ and $0 < r < F$. Such a job will receive exactly the same allocation over the next F time units as it would receive had it been released at a frame boundary. As a result, EDF-tu guarantees the following two key properties.

Property 4.7. *Within any time interval of length F, the processor supply guaranteed to a migrating task τ_i is $u_i \cdot F$. Therefore, within any time interval of length L, the processor supply guaranteed to a migrating task τ_i is at least $\left\lfloor \frac{L}{F} \right\rfloor \cdot u_i \cdot F$.*

Property 4.8. *Within any time interval of length F, the supply guaranteed to the set of fixed tasks on processor s_p is $\sigma_p^f \cdot F$. Therefore, within any time interval of length L, the supply guaranteed to the set of all fixed tasks on processor s_p is at least $\left\lfloor \frac{L}{F} \right\rfloor \cdot \sigma_p^f \cdot F$.*

4.3.3 Optimality

We now show that EDF-tu is HRT-optimal, provided the frame size, F, meets a certain requirement. We also show that EDF-tu is SRT-optimal for any choice of F, with the tardiness of any job being at most F. As F decreases, preemption frequencies increase, so the choice of F is a tradeoff between temporal guarantees and run-time overheads.

4.3.3.1 HRT Optimality

HRT optimality is dealt with in the following theorem.

Theorem 4.6. *If the frame size F divides the periods of all tasks, then all deadlines will be meet.*

Proof. Since the frame size F divides the periods of all tasks, we can represent task periods as

$$T_i = k_i \cdot F, \quad k_i \in \mathbb{Z}^+. \tag{4.42}$$

Migrating tasks. By Property 4.7, within any interval of length T_i, a migrating task τ_i is guaranteed supply of at least $\left\lfloor \frac{T_i}{F} \right\rfloor \cdot u_i \cdot F = k_i \cdot u_i \cdot F = T_i \cdot u_i = C_i$. This implies that no job of any migrating task will miss a deadline.

Fixed tasks. The proof of this case utilizes the following claim.

Claim 4.1. *For real numbers $a, b > 0$ and $x \in \mathbb{Z}^+$, $\left\lfloor \frac{a}{x \cdot b} \right\rfloor \leq \frac{1}{x} \left\lfloor \frac{a}{b} \right\rfloor$.*

Proof. Letting $y = \left\lfloor \frac{a}{b} \right\rfloor$ and $c = a - y \cdot b$, we have $a = y \cdot b + c$, where $y \in \mathbb{Z}$ and $0 \leq c < b$. The latter implies $0 \leq \frac{c}{b} < 1$. Hence, because $x, y \in \mathbb{Z}$, $\left\lfloor \frac{y + \frac{c}{b}}{x} \right\rfloor = \left\lfloor \frac{y}{x} \right\rfloor$. Thus, we have $\frac{1}{x} \left\lfloor \frac{a}{b} \right\rfloor = \frac{y}{x} \geq \left\lfloor \frac{y}{x} \right\rfloor = \left\lfloor \frac{y + \frac{c}{b}}{x} \right\rfloor = \left\lfloor \frac{y \cdot b + c}{x \cdot b} \right\rfloor = \left\lfloor \frac{a}{x \cdot b} \right\rfloor$. □

We now dispense with the case of fixed tasks by contradiction. Let t_d be the first time a job of any fixed task on processor s_p misses its deadline, and let t_0 be the latest time instant before t_d that is idle for fixed tasks on processor s_p, *i.e.*, all jobs of fixed tasks on processor s_p released earlier than t_0 have completed by t_0 and such a job is released at t_0. Within the time interval $[t_0, t_d)$, the demand due to the set of fixed tasks on processor s_p is at most

$$
\begin{aligned}
& \sum_{\substack{\tau_i \text{ is a fixed task} \\ \text{on processor } s_p}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i \\
= \quad & \{\text{by (4.1)}\} \\
& \sum_{\substack{\tau_i \text{ is a fixed task} \\ \text{on processor } s_p}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot T_i \cdot u_i \\
= \quad & \{\text{by (4.42)}\} \\
& \sum_{\substack{\tau_i \text{ is a fixed task} \\ \text{on processor } s_p}} \left\lfloor \frac{t_d - t_0}{k_i \cdot F} \right\rfloor \cdot k_i \cdot F \cdot u_i \\
\leq \quad & \{\text{by Claim 4.1}\} \\
& \sum_{\substack{\tau_i \text{ is a fixed task} \\ \text{on processor } s_p}} \frac{1}{k_i} \cdot \left\lfloor \frac{t_d - t_0}{F} \right\rfloor \cdot k_i \cdot F \cdot u_i \\
= \quad & \{\text{simplifying}\} \\
& \left\lfloor \frac{t_d - t_0}{F} \right\rfloor \cdot F \cdot \sum_{\substack{\tau_i \text{ is a fixed task} \\ \text{on processor } s_p}} u_i
\end{aligned}
$$

$$= \quad \{\text{by (4.28)}\}$$
$$\left\lfloor \frac{t_d - t_0}{F} \right\rfloor \cdot \sigma_p^f \cdot F.$$

By Property 4.8, within the time interval $[t_0, t_d)$, the supply guaranteed to the set of fixed tasks on processor s_p is at least

$$\left\lfloor \frac{t_d - t_0}{F} \right\rfloor \cdot \sigma_p^f \cdot F.$$

This implies that a deadline is not missed at time t_d as assumed. □

By Theorem 4.6, to guarantee HRT optimality, the frame size cannot exceed the greatest common divider (gcd) of all task periods. The gcd could be quite small for some systems (*e.g.*, if at least two periods are relatively prime), yielding high run-time overheads. However, for some systems (*e.g.*, harmonic ones), the frame could be of a reasonable size, yielding acceptable overheads.

4.3.3.2 SRT Optimality

SRT optimality is dealt with in the following theorem.

Theorem 4.7. *Given any frame size $F > 0$, no job will have tardiness exceeding F.*

Proof. As before, we consider migrating and fixed tasks separately.

Migrating tasks. Consider the j^{th} job of the migrating task τ_i, denoted $\tau_{i,j}$. Let t_d be the deadline of $\tau_{i,j}$ and let t_0 be the latest idle instant for task τ_i at or before the release of $\tau_{i,j}$. Also, let t_F be the first time instant at or after t_d such that $t_F - t_0$ is a multiple of F. Then, we have $t_F - t_d < F$.

The number of jobs with deadlines at or before time t_d that τ_i can release at or after time t_0 is at most $\left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \leq \left\lfloor \frac{t_F - t_0}{T_i} \right\rfloor$. The resulting demand is at most $\left\lfloor \frac{t_F - t_0}{T_i} \right\rfloor \cdot C_i \leq (t_F - t_0) \cdot u_i$. Because $(t_F - t_0)$ is a multiple of F, by Prop. 4.7, τ_i is guaranteed a supply of $(t_F - t_0) \cdot u_i$ within $[t_0, t_F)$. This implies that $\tau_{i,j}$ completes by time t_F. Thus, no job of a migrating task will have tardiness exceeding F.

Fixed tasks. Let t_d be the deadline of the j^{th} job $\tau_{i,j}$ of the fixed task τ_i and t_0 be the latest idle instant for fixed tasks on processor s_p at or before the release of $\tau_{i,j}$. Also, let t_F be the first time instant at or after t_d such that $t_F - t_0$ is a multiple of F. Then, we have $t_F - t_d < F$.

The number of jobs with deadlines at or before time t_d that a fixed task τ_k on processor s_p can release at or after time t_0 is at most $\left\lfloor \frac{t_d - t_0}{T_k} \right\rfloor \leq \left\lfloor \frac{t_F - t_0}{T_k} \right\rfloor$, so the total demand due to such jobs is at most

$$\sum_{\substack{\tau_k \text{ is a fixed task} \\ \text{on processor } s_p}} \left\lfloor \frac{t_F - t_0}{T_k} \right\rfloor \cdot C_k \leq (t_F - t_0) \cdot \sum_{\substack{\tau_k \text{ is a fixed task} \\ \text{on processor } s_p}} u_k$$
$$= (t_F - t_0) \cdot \sigma_p^f.$$

Because $(t_F - t_0)$ is a multiple of F, by Prop. 4.8, the fixed tasks on processor p are guaranteed a supply of $(t_F - t_0) \cdot \sigma_p^f$ within $[t_0, t_F)$. This implies that $\tau_{i,j}$ completes by t_F. Thus, no job of a fixed task will have tardiness exceeding F. □

4.3.4 Alternate Assignment Strategies

Given that at most m tasks are migrating under EDF-tu, and these tasks are the heaviest by utilization, two natural questions arise.

- **Q1:** Can we guarantee that fewer than m tasks are migrating?
- **Q2:** Can we require lighter tasks, instead of heavier ones, to migrate?

In this section, we provide counterexamples that show that the answer to each question is no.

Question Q1. We show that the answer to Question Q1 is no by showing that, for any semi-partitioned scheduler, if it is guaranteed that there will be at most k migrating tasks for *any* feasible system, then k cannot be less than m. This result follows from the following counterexample, which consists of m tasks, all of which must migrate.

Example 4.7. Consider a system of m tasks, each with parameters $\tau_i = (1 + \varepsilon, 1)$, where $\varepsilon < 1/m$, to be scheduled on m uniform processors, where $s_1 = 1 + m \cdot \varepsilon$ and $s_i = 1$ for $2 \leq i \leq m$. Conditions (4.2) and (4.3) imply that this system is feasible. Now, if we attempt to assign any single task as fixed, then it must be assigned to processor s_1. However, if we do so, in order to receive enough supply, the remaining $m - 1$ tasks must fully use the remaining residual capacities, *i.e.*, they must fully utilize $m - 1$ processors (s_2 to s_m) and meanwhile also utilize the residual capacity on s_1. Because intra-task parallelism is forbidden, this is infeasible. ◊

The above counterexample shows that we cannot *generally* guarantee that fewer than m tasks will migrate. However, if we examine a *specific* task system, then it may indeed be possible to require fewer than m tasks

to migrate. In fact, in systems that can be fully partitioned, *no* task will migrate. Unfortunately, determining the minimum number of migrating tasks for a specific, concrete task system is NP-hard in the strong sense. This can be shown by transforming from the variable-sized bin-packing problem (Funk, 2004).

Question Q2. The following counterexample shows that the answer to Question Q2 is no as well.

Example 4.8. Consider n tasks to be scheduled on m uniform processors, where $s_1 = 1 + (m+1) \cdot \varepsilon$, where $\varepsilon < (m-1)/(m+1)$, and $s_i = 1$ for $2 \leq i \leq m$. The n tasks include m heavy ones with parameters $(1+\varepsilon, 1)$ and $n-m$ light ones with parameters $(\varepsilon, n-m)$. Conditions (4.2) and (4.3) imply that this system is feasible. Now, if any one of the m heavy tasks is assigned as fixed, then it must be fixed on processor s_1. However, if we do so, the remaining $m-1$ heavy tasks cannot all be allocated shares that match their utilizations without introducing intra-task parallelism. Thus, the remaining system is infeasible. The following is a more formal reasoning for this. ◊

Formal Reasoning for Example 4.8. Let $\psi_{i,p}$ denote the capacity allocated to τ_i on processor s_p, where $0 \leq \psi_{i,p} \leq s_p$. Then, the *portion* of s_p that is allocated to τ_i is

$$\eta_{i,p} = \frac{\psi_{i,p}}{s_p}, \tag{4.43}$$

where $0 \leq \eta_{i,p} \leq 1$. The portion $\eta_{i,p}$ is the needed percentage of CPU time on s_p for τ_i to receive processor supply on s_p that matches its allocated capacity $\psi_{i,p}$ on s_p. Thus, if intra-task parallelism is forbidden, then the following condition must hold for the system to be feasible.

$$\sum_{p=1}^{m} \eta_{i,p} \leq 1, \quad \text{for any } i \tag{4.44}$$

For illustration, consider a task that needs to utilize 70% of the CPU time on one processor and 80% of the CPU time on another processor. This is clearly infeasible if intra-task parallelism is not allowed.

Now, let us examine the specific system in Example 4.8. As shown in Example 4.8, if any one of the m heavy tasks is assigned as fixed, then it must be fixed on processor s_1. Without loss of generality, assume that the heavy task τ_1 is fixed on s_1 and the remaining $m-1$ heavy tasks are $\{\tau_2, \tau_3, \ldots, \tau_m\}$. Since τ_1 is fixed on s_1, the other heavy tasks cannot be allocated shares on s_1 exceeding its residual capacity. Thus,

$$\sum_{i=2}^{m} \psi_{i,1} \leq s_1 - u_1. \tag{4.45}$$

(4.43) and (4.45) imply

$$\sum_{i=2}^{m} \eta_{i,1} \leq 1 - \frac{u_1}{s_1}. \tag{4.46}$$

Moreover, by (4.44), $\sum_{i=2}^{m} \sum_{p=1}^{m} \eta_{i,p} \leq m-1$, *i.e.*, $\sum_{i=2}^{m} \eta_{i,1} + \sum_{i=2}^{m} \sum_{p=2}^{m} \eta_{i,p} \leq m-1$. Therefore,

$$\sum_{i=2}^{m} \sum_{p=2}^{m} \eta_{i,p} \leq (m-1) - \sum_{i=2}^{m} \eta_{i,1}. \tag{4.47}$$

Thus, the allocated shares for the remaining $m-1$ heavy tasks satisfy

$$\begin{aligned}
& \sum_{i=2}^{m} \sum_{p=1}^{m} \psi_{i,p} \\
= \quad & \{\text{by (4.43)}\} \\
& \sum_{i=2}^{m} \sum_{p=1}^{m} \eta_{i,p} \cdot s_p \\
= \quad & \{\text{rearranging and by } s_p = 1 \text{ for } 2 \leq p \leq m \text{ as in Example 4.8}\} \\
& \sum_{i=2}^{m} \eta_{i,1} \cdot s_1 + \sum_{i=2}^{m} \sum_{p=2}^{m} \eta_{i,p} \cdot 1 \\
\leq \quad & \{\text{by (4.47)}\} \\
& \sum_{i=2}^{m} \eta_{i,1} \cdot s_1 + (m-1) - \sum_{i=2}^{m} \eta_{i,1} \\
= \quad & \{\text{rearranging}\} \\
& (m-1) + \sum_{i=2}^{m} \eta_{i,1} \cdot (s_1 - 1) \\
\leq \quad & \{\text{by (4.46)}\} \\
& (m-1) + (1 - \frac{u_1}{s_1}) \cdot (s_1 - 1) \\
= \quad & \{\text{by the definitions of } s_1 \text{ and } u_1 \text{ in Example 4.8}\} \\
& (m-1) + \left(1 - \frac{1+\varepsilon}{1+(m+1)\cdot\varepsilon}\right) \cdot (1+(m+1)\cdot\varepsilon - 1) \\
= \quad & \{\text{simplifying}\} \\
& (m-1) + \frac{m \cdot \varepsilon}{1+(m+1)\cdot\varepsilon} \cdot (m+1) \cdot \varepsilon \\
= \quad & \{\text{simplifying}\}
\end{aligned}$$

$$
\begin{aligned}
& (m-1)+\frac{m\cdot(m+1)\cdot\varepsilon}{1+(m+1)\cdot\varepsilon}\cdot\varepsilon \\
= \quad & \{\text{simplifying}\} \\
& (m-1)+\frac{m}{\frac{1}{(m+1)\cdot\varepsilon}+1}\cdot\varepsilon \\
< \quad & \{\text{as stated in Example 4.8, } \varepsilon < (m-1)/(m+1)\} \\
& (m-1)+\frac{m}{\frac{1}{(m+1)\cdot(m-1)/(m+1)}+1}\cdot\varepsilon \\
= \quad & \{\text{simplifying}\} \\
& (m-1)+\frac{m}{\frac{1}{m-1}+1}\cdot\varepsilon \\
= \quad & \{\text{simplifying}\} \\
& (m-1)+(m-1)\cdot\varepsilon,
\end{aligned}
$$

which is the needed total share allocation of the remaining $m-1$ heavy tasks. Thus, the remaining system is not feasible if intra-task parallelism is forbidden.

4.3.5 Evaluation

The frame size F used in EDF-tu is a tunable parameter. For any feasible task system, tardiness will always be at most F, and if F is set low enough, tardiness will be zero. Given this, it would not be very interesting to experimentally examine issues related to schedulability. However, for a given task system, the Level-Algorithm-induced preemption pattern within a frame is the same regardless of its size, and preemption frequencies over time are higher when F is smaller. Thus, it *is* interesting to experimentally evaluate the number of tasks that are required to migrate and the number of preemptions experienced by such tasks, as it is these tasks that give rise to preemptions induced by the Level Algorithm. In this section, we briefly discuss an experimental evaluation that focuses on these two metrics.

Experimental setup. We assessed the impact of both metrics by randomly generating feasible task systems and determining for each generated system the number of migrating tasks and the number of preemptions experienced by such tasks per frame. In experimental studies that focus on identical platforms, choosing an overall utilization cap implicitly defines the considered multiprocessor platform. However, in the uniform case, processor speeds must be selected, and the number of such speed settings is unbounded for a given total

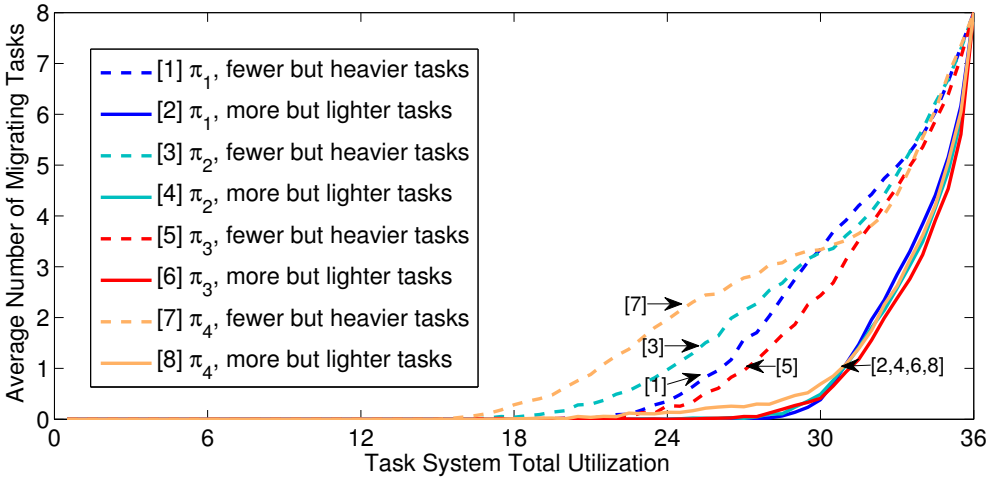


Figure 4.10: Number of migrating tasks.

utilization. To reasonably constrain our experiments, we considered systems of eight processors with a total processor capacity of 36. We considered four such platforms, with speeds as follows: $\pi_1 = \{6, 6, 6, 6, 3, 3, 3, 3\}$, $\pi_2 = \{8, 8, 4, 4, 4, 4, 2, 2\}$, $\pi_3 = \{8, 7, 6, 5, 4, 3, 2, 1\}$, and $\pi_4 = \{15, 3, 3, 3, 3, 3, 3, 3\}$. We used the same framework as described in Section 4.2.3 to randomly generate feasible task systems. When using this framework, two categories of task systems are generated: systems that have fewer tasks but heavier tasks (by utilization), and systems that have more tasks but lighter tasks.

For each platform and task generating pattern, we varied total utilization within $[0, 36]$ by increments of 0.5, and for each total utilization, we generated 1,000 feasible task systems. Figure 4.10 plots the average number of migrating tasks required for each such set of 1,000 task systems. For every generated task system, we also simulated EDF-tu and recorded the maximum number of preemptions per frame of any migrating task. Figure 4.11 plots the average of these maximum values for each set of 1,000 task systems.

Results. As seen in Figures 4.10 and 4.11, the number of migrating tasks is often modest, and these tasks often experience only a moderate number of preemptions per frame. With total utilization as high as 30, the number of migrating tasks (on average) typically is at most four, and the number of preemptions per frame (on average) is at most five. Even in the extreme case that the total utilization achieves the total speed of the platform, the number of preemptions per frame (on average) is still less than 25. While 25 preemptions per frame may seem somewhat high, recall that in a SRT system, we can define the frame size F to be quite large at the cost of increasing the tardiness bound.

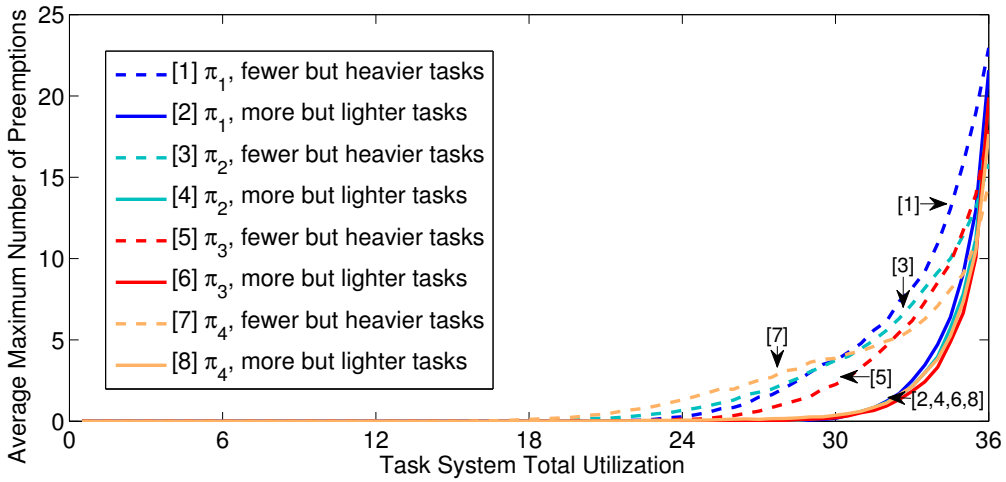


Figure 4.11: Maximum number of preemptions of migrating tasks per frame.

4.4 Chapter Summary

In this chapter, we have presented two EDF-based semi-partitioned scheduling algorithms for uniform multiprocessors. The first one, EDF-sh, provides SRT guarantees only and it is not SRT-optimal, *i.e.*, some feasible systems are not SRT-schedulable under EDF-sh. Nonetheless, the total utilization of an SRT-schedulable system under EDF-sh can be as high as the total platform capacity. Furthermore, EDF-sh restricts task migrations to occur at job boundaries only, *i.e.*, no job, even of a migrating task, migrates under EDF-sh. In contrast, the second algorithm, EDF-tu, is SRT-optimal and includes a tunable parameter, called frame size. For any positive value of the frame size, tardiness is guaranteed to be at most this value. Furthermore, if the frame size divides all task periods, EDF-sh becomes HRT-optimal and ensures zero tardiness.

For scheduling n tasks on m uniform processors, EDF-sh allows at most $m-1$ tasks to migrate while EDF-tu allows m. Interestingly, we have given a feasible system where at least m task must migrate, or unbounded tardiness may be inevitable. This shows that, m as the maximum number of migrating tasks under EDF-tu is tight for any HRT- or SRT-optimal scheduler. This also explains the lack of optimality for EDF-sh.

CHAPTER 5: ALLOWING INTRA-TASK PARALLELISM ON UNIFORM PLATFORMS[1]

In this chapter, we continue to focus on the uniform multiprocessor model. However, in contrast to the prior two chapters, which focused on the conventional sporadic task model, we shift our attention to the *npc-sporadic* task model here. The conventional sporadic task model restricts each *task* to execute sequentially, which might compromise the actual parallelism potential of the system. For example, many detection algorithms in computer vision, such as the *Histogram of Oriented Gradients* (HOG) algorithm for recognizing pedestrians (Dalal and Triggs, 2005), may be performed on each frame of a video independently. Modeling computation like that by the conventional sporadic task model results in implicitly ruling out the possibility of processing consecutive frames simultaneously, which could leave a multiprocessor platform unnecessarily under-utilized. Thus, it could be beneficial to model such a workload by a less-restrictive task model—the npc-sporadic task model, which allows consecutive jobs of the same task to execute in parallel.

We consider the scheduling of npc-sporadic task systems on a uniform multiprocessor platform under both preemptive and non-preemptive G-EDF. We show that both of these algorithms guarantee bounded job response times for any feasible npc-sporadic task system. Interestingly, the SRT-feasibility condition we establish for npc-sporadic tasks differs from that for sporadic tasks in Section 3.2. Of the two algorithms we consider, preemptive G-EDF is more greedy in executing jobs on faster processors, and therefore ensures a better response-time bound; in contrast, non-preemptive G-EDF only ensures a looser response-time bound, but does not migrate jobs away from slower processors when faster processors becomes available; this characteristic could lead to energy savings.

Organization. In the rest of this chapter, we describe the considered platforms and task systems more carefully (Section 5.1), provide proof-specific preliminaries (Section 5.2), prove our response-time bounds for preemptive (Section 5.3) and non-preemptive (Section 5.4) G-EDF, and present an experimental evaluation (Section 5.5).

[1]Contents of this chapter previously appeared in preliminary form in the following paper:

Yang, K. and Anderson, J. (2014a). Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *Proceedings of the 12th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 30–39.

5.1 System Model

In this chapter, we consider a uniform multiprocessor platform consisting of m processors. Processor i, where $1 \leq i \leq m$, has an associated *speed* denoted by a real number s_i, which represents the amount of work that can be done on this processor within one time unit. We also identify each processor by its speed and assume the processors are decreasingly ordered by their speeds, *i.e.*, we denote a uniform multiprocessor platform by $\pi = \{s_1, s_2, \ldots, s_m\}$ where $s_i \geq s_{i+1}$ for $i = 1, 2, \ldots, m-1$. The cumulative speed of the i fastest processors is $S_i = \sum_{k=1}^{i} s_k$. Also, we assume time is continuous.

A *task* is a sequential piece of code and a *job* is an instance (or an invocation) of a task. We let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote the task set to be scheduled. Each task τ_i is characterized by (C_i, T_i, D_i), where C_i is τ_i's *worst-case execution time* (WCET) on a unit-speed processor, T_i is its *period* (the minimum separation of any two jobs), and D_i is its *relative deadline*. $\tau_{i,j}$ is the j^{th} job of τ_i. The release time of $\tau_{i,j}$ is $r_{i,j}$ and its absolute deadline is $d_{i,j}$, where $d_{i,j} = r_{i,j} + D_i$.

On a uniform multiprocessor, C_i, called the *worst-case execution requirement* of τ_i, is defined relative to a processor of speed 1.0. Thus, the WCET of τ_i when entirely executing on processor s_p is $\frac{C_i}{s_p}$ $(1 \leq p \leq m)$. We let $C_{max} = \max\{C_i \mid 1 \leq i \leq n\}$. Note that, the *execution requirement* of a job may differ from its *execution time* if non-unit-speed processor exists. In prior work on identical multiprocessors, the two are the same, since speeds are usually normalized to 1.0. The *utilization* of a task τ_i is $u_i = \frac{C_i}{T_i}$. The total utilization of τ is $U_\tau = \sum_{i=1}^{n} u_i$. We also use τ to refer to the set of all jobs generated by tasks in τ.

In this chapter, we consider npc-sporadic tasks, which have no intra-task precedence constraints. The main difference between the conventional sporadic task model and the npc-sporadic task model is that the former requires successive jobs of each task to execute in sequence while the latter allows them to execute in parallel. That is, in the conventional sporadic task model, job $\tau_{i,j+1}$ cannot commence execution until its predecessor $\tau_{i,j}$ completes, even if $r_{i,j+1}$, the release time of $\tau_{i,j+1}$, has elapsed; in contrast, in npc-sporadic task model, any job can execute as soon as it is released. Additionally, in the npc-sporadic model, a task is allowed to have a utilization greater than the fastest processor's speed. Note that, although we allow intra-task parallelism, each individual job still must execute sequentially.

Feasibility conditions.

On uniform multiprocessors, Funk et al. (2001) derived the following feasibility condition for conventional HRT implicit-deadline (*i.e.*, $D_i = T_i$) *periodic* task systems. Let U_i denote the sum of the largest

i utilizations in τ and assume the number of tasks n is at least the number of processors m. Then, an implicit-deadline periodic task set τ is HRT-feasible on a uniform multiprocessor π if and only if

$$U_\tau \leq S_m, \tag{5.1}$$

and

$$U_i \leq S_i, \text{ for } i = 1, 2, \cdots, m-1. \tag{5.2}$$

As shown in Section 3.2, this is also a necessary and sufficient feasibility condition for conventional implicit-deadline *sporadic* task systems in *both* HRT and SRT senses. The former requires all deadlines to be met while the latter only requires response times to be bounded.

In the npc-sporadic task model, where intra-task precedence constraints are eliminated, it is clear that (5.1) and (5.2) are also a necessary and sufficient feasibility condition for HRT implicit-deadline task systems, since in such systems, every job has to complete before its successor is released and hence there is no difference between conventional sporadic task systems and npc-sporadic ones. However, for SRT systems, (5.2) is not required, as we will show that response-time bounds do not rely on (5.2). Furthermore, for arbitrary-deadline tasks, our analysis applies as well. On the other hand, (5.1) is always required. A violation of (5.1) means the system is overutilized, and therefore response times will increase without bound. To summarize, an npc-sporadic task system τ is SRT-feasible on π if and only if (5.1) holds.

5.2 Preliminaries

Definition 5.1. At a time instant t, a job $\tau_{i,j}$ is *unreleased* if $t < r_{i,j}$, *pending* if $t \geq r_{i,j}$ and $\tau_{i,j}$ has not completed execution by t, and *complete* if $\tau_{i,j}$ has completed by t.

Definition 5.2. We let $\mathsf{A}(\mathcal{S}, \tau_{i,j}, t_1, t_2)$ denote the cumulative processor capacity allocation to job $\tau_{i,j}$ in an arbitrary schedule $\mathcal{S}$ within the time interval $[t_1, t_2]$.

Also, we let $\mathsf{A}(\mathcal{S}, \mathcal{J}, t_1, t_2)$ denote the cumulative processor capacity allocation to the jobs in job set $\mathcal{J}$ in an arbitrary schedule $\mathcal{S}$ within the time interval $[t_1, t_2]$, *i.e.*,

$$\mathsf{A}(\mathcal{S}, \mathcal{J}, t_1, t_2) = \sum_{\tau_{i,j} \in \mathcal{J}} \mathsf{A}(\mathcal{S}, \tau_{i,j}, t_1, t_2). \tag{5.3}$$

Ideal schedule. We let $\pi_{\text{IDEAL}} = \{u_1, u_2, \ldots, u_n\}$ denote an ideal multiprocessor for the task set τ, where π_{IDEAL} consists of n processors with speeds that exactly match the utilizations of the n tasks in τ, respectively. Let $\mathcal{I}$ be the partitioned schedule for τ on π_{IDEAL}, where each task τ_i in τ is assigned to the processor of speed u_i. Then, in $\mathcal{I}$, every job in τ commences execution at its release time and completes execution within one period (it exactly executes for one period if and only if its actual execution requirement matches its worst-case execution requirement).

Thus, we have, $\mathsf{A}(\mathcal{I}, \tau_{i,j}, t_1, t_2) \leq u_i \cdot (t_2 - t_1)$, and for an arbitrary job set $\mathcal{J} \subseteq \tau$,

$$\mathsf{A}(\mathcal{I}, \mathcal{J}, t_1, t_2) \leq U_\tau \cdot (t_2 - t_1). \tag{5.4}$$

This is similar to the processor sharing (PS) schedule considered in prior work with respect to identical multiprocessors (Devi and Anderson, 2008). However, the above notion of an ideal schedule is preferable with respect to uniform multiprocessors, because it clearly prevents a single job from executing in parallel with itself.

Note that, in the ideal schedule, a constrained-deadline task τ_i (*i.e.*, $D_i < T_i$) may not complete execution at its deadline. The following definition gives an upper bound on the amount of work that completes later than its deadline in the ideal schedule.

Definition 5.3. Let $L_i = u_i \cdot \max\{0, T_i - D_i\}$, and let $L_\tau = \sum_{i=1}^{n} L_i$. Then, in $\mathcal{I}$, at any time instant t, the amount of incomplete work with deadline at or before t is at most L_τ.

Definition 5.4. We denote the difference between the allocation to a job $\tau_{i,j}$ in $\mathcal{I}$ and in a schedule $\mathcal{S}$ within $[0,t]$ as

$$\mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) = \mathsf{A}(\mathcal{I}, \tau_{i,j}, 0, t) - \mathsf{A}(\mathcal{S}, \tau_{i,j}, 0, t), \tag{5.5}$$

and such an allocation difference for an arbitrary job set $\mathcal{J}$ is

$$\mathsf{LAG}(\mathcal{J}, t, \mathcal{S}) = \sum_{\tau_{i,j} \in \mathcal{J}} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}). \tag{5.6}$$

By (5.5) and Definition 5.2, for any time interval $[t_1, t_2]$ we have

$$\mathsf{lag}(\tau_{i,j}, t_2, \mathcal{S}) = \mathsf{lag}(\tau_{i,j}, t_1, \mathcal{S}) + \mathsf{A}(\mathcal{I}, \tau_{i,j}, t_1, t_2) - \mathsf{A}(\mathcal{S}, \tau_{i,j}, t_1, t_2); \tag{5.7}$$

and by (5.3), (5.6), and (5.7),

$$\mathsf{LAG}(\mathcal{J},t_2,\mathcal{S}) = \mathsf{LAG}(\mathcal{J},t_1,\mathcal{S}) + \mathsf{A}(\mathcal{I},\mathcal{J},t_1,t_2) - \mathsf{A}(\mathcal{S},\mathcal{J},t_1,t_2). \tag{5.8}$$

Lemma 5.1. *If a job $\tau_{i,j}$ is unreleased or complete at t, then $\mathsf{lag}(\tau_{i,j},t,\mathcal{S}) \leq 0$; if $\tau_{i,j}$ is pending at t, then $\mathsf{lag}(\tau_{i,j},t,\mathcal{S}) \leq C_i$.*

Proof. Follows immediately from Definitions 5.1 and 5.4. □

Job of interest. To derive response-time bounds, we consider an arbitrary job $\tau_{k,l}$ in τ, and upper bound its response time. Let t_d be the absolute deadline of $\tau_{k,l}$, *i.e.*, $t_d = d_{k,l}$.

Definition 5.5. In the rest of this chapter, we let Ψ be the job set consisting of all jobs with deadlines at or before t_d. The jobs in Ψ are called *competing jobs* for $\tau_{k,l}$. At time instant t, the total incomplete work due to all jobs in Ψ is called *competing work* at t.

Definition 5.6. If at a time instant t, all of the m processors are executing jobs in Ψ, then t is a *busy instant* for Ψ; otherwise t is a *non-busy instant* for Ψ. If in the time interval $[t_1,t_2]$ every time instant is a busy instant for Ψ, then $[t_1,t_2]$ is a *busy interval* for Ψ.

Lemma 5.2. *If in $\mathcal{S}$, $[t_1,t_2]$ is a busy interval for Ψ, then $\mathsf{LAG}(\Psi,t_1,\mathcal{S}) \geq \mathsf{LAG}(\Psi,t_2,\mathcal{S})$.*

Proof. Follows from the previous definitions. □

Definition 5.7. For any time instant t, we let t^+ denote the time instant $(t+\varepsilon)$ and we let t^- denote the time instant $(t-\varepsilon)$, where $\varepsilon \to 0^+$.

5.3 Response-Time Bounds under Preemptive G-EDF

In this section, we consider the preemptive G-EDF scheduler that works as follows.

- At any time instant, if there are at most m pending jobs, then all of them are scheduled; if there are more than m pending jobs, then the m such jobs with the earliest deadlines are scheduled. Deadline ties are broken arbitrarily.
- For any two jobs $\tau_{i,j}$ scheduled on processor s_p and $\tau_{a,b}$ scheduled on processor s_q, where $p < q$, we have $d_{i,j} \leq d_{a,b}$ (note that, given how we order processors, $s_p \geq s_q$).

Moreover, in this section, we let $\mathcal{S}$ denote a preemptive G-EDF schedule of τ on π.

5.3.1 Basic Bounds

We first present a proof to derive basic response-time bounds. This proof is more similar to the SRT analysis framework for G-EDF by Devi and Anderson (2008), and therefore is easier to understand. We will improve the basic bounds proved here in Section 5.3.2. Recall that $\tau_{k,l}$ is the analyzed job.

Lemma 5.3. *At any non-busy instant t at or before t_d, $\mathsf{LAG}(\Psi, t, \mathcal{S}) \leq (m-1) \cdot C_{max}$.*

Proof. We decompose Ψ into three disjoint subsets: Ψ_1, Ψ_2, and Ψ_3 consisting of jobs that are *unreleased*, *pending*, and *complete*, respectively. Since in the npc-sporadic task model intra-task precedence constraints are removed, under preemptive G-EDF, there can be at most $(m-1)$ jobs in Ψ that are pending at t, *i.e.*, $|\Psi_2| \leq m-1$. Thus,

$$
\begin{aligned}
& \mathsf{LAG}(\Psi, \mathcal{S}, t) \\
= & \{\text{by the definition of } \Psi_1, \Psi_2, \text{ and } \Psi_3\} \\
& \mathsf{LAG}(\Psi_1, t, \mathcal{S}) + \mathsf{LAG}(\Psi_2, t, \mathcal{S}) + \mathsf{LAG}(\Psi_3, t, \mathcal{S}) \\
= & \{ \text{ by (5.6) } \} \\
& \sum_{\tau_{i,j} \in \Psi_1} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) + \sum_{\tau_{i,j} \in \Psi_2} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) + \sum_{\tau_{i,j} \in \Psi_3} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) \\
\leq & \{ \text{ by Lemma 5.1 } \} \\
& \sum_{\tau_{i,j} \in \Psi_1} 0 + \sum_{\tau_{i,j} \in \Psi_2} C_i + \sum_{\tau_{i,j} \in \Psi_3} 0 \\
\leq & |\Psi_2| \cdot C_{max} \\
\leq & (m-1) \cdot C_{max}.
\end{aligned}
$$

The lemma follow. □

Lemma 5.4. *After t_d, once $\tau_{k,l}$ executes, it will continuously execute until it completes.*

Proof. After t_d, no job with deadline earlier than t_d can be released, *i.e.*, no job that can preempt $\tau_{k,l}$ can be released. Thus, once $\tau_{k,l}$ executes, it will continually execute until it completes, though it could migrate among processors. □

Lemma 5.5. *In $\mathcal{S}$, the competing work for $\tau_{k,l}$ at t_d is at most $L_\tau + (m-1)\cdot C_{max}$.*

Proof. By Definitions 5.3 and 5.4, the competing work pending at t_d in $\mathcal{S}$ is at most $L_\tau + \mathsf{LAG}(\Psi, \mathcal{S}, t_d)$. Let t' be the latest non-busy instant at or before t_d (or time 0 if no such non-busy instant exists). Then, by Lemma 5.2, $\mathsf{LAG}(\Psi, t', \mathcal{S}) \geq \mathsf{LAG}(\Psi, t_d, \mathcal{S})$. Also, by Lemma 5.3, $\mathsf{LAG}(\Psi, t', \mathcal{S}) \leq (m-1)\cdot C_{max}$. Thus, $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) \leq (m-1)\cdot C_{max}$ and therefore the lemma follows. □

Lemma 5.6. *Let W be the competing work for $\tau_{k,l}$ at t_d. Then the job of interest, $\tau_{k,l}$, will complete execution no later than time*

$$t_d + \frac{W - C_k}{S_m} + \frac{C_k}{s_m}.$$

Proof. Suppose that $\tau_{k,l}$ is not complete at or before t_d. Let δ be the amount of work of $\tau_{k,l}$ that has been completed by t_d and $e_{k,l}$ be the real execution requirement of $\tau_{k,l}$. Then the remaining execution work of $\tau_{k,l}$ at t_d is $e_{k,l} - \delta$. If $\tau_{k,l}$ does not execute within $[t_d, t_d + \frac{W-(e_{k,l}-\delta)}{S_m})$, then $[t_d, t_d + \frac{W-(e_{k,l}-\delta)}{S_m})$ must be a busy interval for Ψ. In this case, the competing work that is completed within $[t_d, t_d + \frac{W-(e_{k,l}-\delta)}{S_m})$ is $W - (e_{k,l} - \delta)$ (since within a busy interval, all processors execute competing work and the total speed is S_m), and the remaining competing work at $t_d + \frac{W-(e_{k,l}-\delta)}{S_m}$ is $e_{k,l} - \delta$, which must be totally due to $\tau_{k,l}$. Therefore, $\tau_{k,l}$ will execute at time $t_d + \frac{W-(e_{k,l}-\delta)}{S_m}$. Thus, if $\tau_{k,l}$ is not complete at or before t_d, then the latest time when $\tau_{k,l}$ commences execution after t_d is $t_d + \frac{W-(e_{k,l}-\delta)}{S_m}$. By Lemma 5.4, $\tau_{k,l}$ will not be preempted once it executes after t_d. Also, since the minimum execution speed is s_m, $\tau_{k,l}$ will complete within $\frac{e_{k,l}-\delta}{s_m}$ time units. Therefore, $\tau_{k,l}$ will complete by

$$\begin{aligned}
& t_d + \frac{W - (e_{k,l} - \delta)}{S_m} + \frac{e_{k,l} - \delta}{s_m} \\
=& \{\text{rearranging}\} \\
& t_d + \frac{W}{S_m} - \frac{e_{k,l}}{S_m} + \frac{e_{k,l}}{s_m} + (\frac{\delta}{S_m} - \frac{\delta}{s_m}) \\
\leq& \{\text{since } \delta \geq 0 \text{ and } S_m \geq s_m\} \\
& t_d + \frac{W}{S_m} - \frac{e_{k,l}}{S_m} + \frac{e_{k,l}}{s_m} \\
\leq& \{\text{since } e_{k,l} \leq C_k \text{ and } S_m \geq s_m\} \\
& t_d + \frac{W}{S_m} - \frac{C_k}{S_m} + \frac{C_k}{s_m} \\
=& t_d + \frac{W - C_k}{S_m} + \frac{C_k}{s_m}.
\end{aligned}$$

The lemma follows. □

Theorem 5.1. *The response time of an arbitrary job $\tau_{k,l}$ in τ under preemptive G-EDF scheduling on π is at most*

$$D_k + \frac{L_\tau + (m-1)\cdot C_{max} - C_k}{S_m} + \frac{C_k}{s_m}.$$

Proof. Follows from Lemmas 5.5 and 5.6. □

5.3.2 Improved Bounds

We now show that the response-time bound above can be improved in several ways.

First, we can derive a better bound on the LAG at t_d or even an arbitrary time instant t by considering *LAG non-increasing intervals* instead of busy intervals.

Definition 5.8. We introduce an integer Λ such that $S_{\Lambda-1} < U_\tau$ and $S_\Lambda \geq U_\tau$ ($1 \leq \Lambda \leq m$). If at time instant t, at least Λ processors are executing jobs in Ψ, then t is a *LAG non-increasing instant*. If in the time interval $[t_1, t_2]$ every time instant is a LAG non-increasing instant, then $[t_1, t_2]$ is a *LAG non-increasing interval* for Ψ.

By the rules of preemptive G-EDF, it is clear that at any time instant, if p processors ($1 \leq p \leq m$) execute jobs in Ψ, then they must be the p fastest ones. This property ensures that LAG for Ψ cannot increase within a LAG non-increasing interval. Then, we can derive following lemma.

Lemma 5.7. *For any time instant t, $\mathsf{LAG}(\Psi, t, \mathcal{S}) \leq (\Lambda - 1)\cdot C_{max}$.*

Proof. This proof is similar to Lemma 5.5. We instead consider the latest time instant that is not a LAG non-increasing instant at or before t. The definition of LAG non-increasing instant and the property in the prior paragraph ensure counterparts for Lemmas 5.3 and 5.2, respectively. Thus, the lemma follows. □

Furthermore, in Section 5.3.1, we only considered the execution of $\tau_{k,l}$ after t_d, and we pessimistically assumed $\tau_{k,l}$ is executed at the minimum speed s_m. Actually, we can consider the execution of $\tau_{k,l}$ as early as it is released, and derive several linear constraints, and solve a corresponding linear program. The following lemma shows this.

Definition 5.9. As defined in prior work (Funk, 2004), the *identicalness* of the multiprocessor platform π is

$$\lambda = \max_{1 \leq i \leq m-1} \left\{ \frac{s_{i+1} + s_{i+2} + \cdots + s_m}{s_i} \right\}.$$

That is,

$$\lambda = \max_{1 \le i \le m-1} \left\{ \frac{S_m - S_i}{s_i} \right\}. \tag{5.9}$$

Note that, $\lambda \le m-1$. Also, $\lambda = m-1$ if and only if π is an identical multiprocessor.

Lemma 5.8. *Suppose the competing work for $\tau_{k,l}$ at $r_{k,l}$ is W. Then the response time of $\tau_{k,l}$ is upper bounded by*

$$\frac{W}{S_m} + \frac{\lambda}{S_m} C_k.$$

Proof. In the time interval between $r_{k,l}$ and $\tau_{k,l}$'s completion, let x_0 denote the cumulative time in which $\tau_{k,l}$ is not executing, and let x_i $(1 \le i \le m)$ denote the cumulative time in which $\tau_{k,l}$ is executing on processor s_i. Then, the response time of $\tau_{k,l}$ is $\sum_{i=0}^{m} x_i$.

Since we cannot execute $\tau_{k,l}$ for more than its worst-case execution requirement, we have the linear constraint

$$\sum_{i=1}^{m} s_i x_i \le C_k.$$

By the rules of preemptive G-EDF, after $r_{k,l}$, when $\tau_{k,l}$ is not complete and is not currently executing, all of the m processors must execute jobs in Ψ; and when $\tau_{k,l}$ is executing on processor s_i, the fastest i processors, *i.e.*, s_1 to s_i, must execute jobs in Ψ. Since W is the competing work at $r_{k,l}$, the execution of jobs in Ψ after $r_{k,l}$ cannot exceed W. Therefore, we have the linear constraint

$$S_m x_0 + \sum_{i=1}^{m} S_i x_i \le W.$$

We now manually solve this linear programming problem by the Simplex Algorithm (Dantzig, 1998), assuming that C_k, W, s_i, and S_i $(1 \le i \le m)$ are constants and each x_i $(0 \le i \le m)$ is a variable. To do so, we introduce two auxiliary variables, x_{m+1} and x_{m+2}, to rewrite this problem in slack form. Specifically, we maximize

$$z = \sum_{i=0}^{m} x_i,$$

subject to

$$\begin{cases} x_{m+1} = C_k - \sum_{i=1}^{m} s_i x_i, \\ x_{m+2} = W - S_m x_0 - \sum_{i=1}^{m} S_i x_i, \\ x_0, x_1, x_2, \cdots, x_{m+2} \geq 0. \end{cases}$$

First, we pivot x_0 with x_{m+2}. Then, in the resulting program, we pivot x_h, where h satisfies $\frac{S_m - S_h}{s_h} = \max_{1 \leq i \leq m} \{\frac{S_m - S_i}{s_i}\} = \lambda$, with x_{m+1}. The final program is to maximize

$$z = \sum_{1 \leq i \leq m \wedge i \neq h} \left(\left(\frac{S_m - S_i}{s_i} \right) - \left(\frac{S_m - S_h}{s_h} \right) \right) \frac{s_i}{S_m} x_i - \left(1 - \frac{S_h}{S_m} \right) \frac{x_{m+1}}{s_h} - \frac{x_{m+2}}{S_m} + \frac{W}{S_m} + \left(1 - \frac{S_h}{S_m} \right) \frac{C_k}{s_h}, \tag{5.10}$$

subject to

$$\begin{cases} x_h = \frac{C_k}{s_h} - \sum_{1 \leq i \leq m \wedge i \neq h} \frac{s_i}{s_h} x_i - \frac{x_{m+1}}{s_h}, \\ x_0 = \frac{W}{S_m} - \sum_{1 \leq i \leq m \wedge i \neq h} \frac{S_i}{S_m} x_i - \frac{x_{m+2}}{S_m} - \frac{S_h}{S_m} \left(\frac{C_k}{s_h} - \sum_{1 \leq i \leq m \wedge i \neq h} \frac{s_i}{s_h} x_i - \frac{x_{m+1}}{s_h} \right), \\ x_0, x_1, x_2, \cdots, x_{m+2} \geq 0. \end{cases}$$

By the definition of h, all the coefficients of the x terms of z in (5.10) are negative or zero. Therefore, when $x_h = \frac{C_k}{s_h}$, $x_0 = (\frac{W}{S_m} - \frac{S_h}{S_m} \cdot \frac{C_k}{s_h})$, and $x_i = 0$ (for all $i \neq 0$ and $i \neq h$), z has its maximum value, which is

$$\begin{aligned} z_{max} &= \frac{W}{S_m} + \frac{S_m - S_h}{S_m s_h} C_k \\ &= \{\text{by (5.9) and by the definition of } h\} \\ &\quad \frac{W}{S_m} + \frac{\lambda}{S_m} C_k. \end{aligned}$$

The lemma follows. □

Next, we upper bound the competing work for $\tau_{k,l}$ at $r_{k,l}$ by the following lemma.

Lemma 5.9. *The competing work for $\tau_{k,l}$ at $r_{k,l}$ is at most*

$$U_\tau \cdot D_k + L_\tau + (\Lambda - 1) \cdot C_{max}.$$

Proof. By Definitions 5.3, 5.4, and 5.5, the competing work for $\tau_{k,l}$ at $r_{k,l}$ is at most

$$
\begin{aligned}
& L_\tau + \mathsf{A}(\mathcal{I}, \Psi, 0, t_d) - \mathsf{A}(\mathcal{S}, \Psi, 0, r_{k,l}) \\
= & \{\text{by Definition 5.2}\} \\
& L_\tau + \mathsf{A}(\mathcal{I}, \Psi, 0, r_{k,l}) + \mathsf{A}(\mathcal{I}, \Psi, r_{k,l}, t_d) - \mathsf{A}(\mathcal{S}, \Psi, 0, r_{k,l}) \\
= & \{\text{by (5.5) and (5.6)}\} \\
& \mathsf{A}(\mathcal{I}, \Psi, r_{k,l}, t_d) + L_\tau + \mathsf{LAG}(\Psi, r_{k,l}, \mathcal{S}) \\
\leq & \{\text{by (5.4)}\} \\
& U_\tau \cdot (t_d - r_{k,l}) + L_\tau + \mathsf{LAG}(\Psi, r_{k,l}, \mathcal{S}) \\
= & \{\text{since } t_d = d_{k,l} = r_{k,l} + D_k \text{ and by Lemma 5.7}\} \\
& U_\tau \cdot D_k + L_\tau + (\Lambda - 1) \cdot C_{max}.
\end{aligned}
$$

The lemma follows. □

Theorem 5.2. *The response time of an arbitrary job $\tau_{k,l}$ in τ under preemptive G-EDF scheduling on π is at most*

$$\frac{U_\tau}{S_m} \cdot D_k + \frac{1}{S_m} \cdot L_\tau + \frac{(\Lambda - 1)}{S_m} \cdot C_{max} + \frac{\lambda}{S_m} C_k.$$

Proof. Follows from Lemmas 5.8 and 5.9. □

5.4 Response-Time Bounds under Non-Preemptive G-EDF

The non-preemptive G-EDF scheduler is similar to the preemptive G-EDF scheduler, except that once a job is selected for execution, it runs to completion without preemption or migration. Also, we do not require faster processors to be favored when scheduling jobs; instead, we can always favor slower ones for energy efficiency (if desired). In this section, we let $\mathcal{S}$ be the non-preemptive G-EDF schedule of τ on π.

Definition 5.10. Suppose time instant t is a non-busy time instant for Ψ. If every pending job in Ψ is currently executing, then t is a *non-blocking non-busy instant* for Ψ; otherwise (*i.e.*, some pending job in Ψ is blocked by jobs that are not in Ψ), t is a *blocking non-busy instant* for Ψ. If in a time interval $[t_1, t_2]$ every time instant is a blocking non-busy instant for Ψ, then $[t_1, t_2]$ is a *blocking non-busy interval* for Ψ.

Definition 5.11. The set of jobs that are not in Ψ but currently executing in schedule $\mathcal{S}$ at time instant t is denoted $\mathcal{B}(t)$ and the incomplete work of jobs in $\mathcal{B}(t)$ is denoted $\mathsf{B}(t)$, called *blocking work*.

It is clear that $|\mathcal{B}(t)| \leq m$ for all t, and therefore $\mathsf{B}(t) \leq m \cdot C_{max}$ for all t.

5.4.1 Basic Bounds

As before, we first derive basic response-time bounds. We will improve the basic bounds in Section 5.4.2.

Lemma 5.10. *At any non-blocking non-busy instant t, $\mathsf{LAG}(\Psi,\mathcal{S},t) + \mathsf{B}(t) \leq m \cdot C_{max}$.*

Proof. Let $b = |\mathcal{B}(t)|$ $(0 \leq b \leq m)$. Then $\mathsf{B}(t) \leq b \cdot C_{max}$, and by the definition of a non-blocking non-busy instant, the number of pending jobs in Ψ is at most $m - b$. Similarly to Lemma 5.3, we have $\mathsf{LAG}(\Psi,\mathcal{S},t) \leq (m-b) \cdot C_{max}$. Thus, $\mathsf{LAG}(\Psi,\mathcal{S},t) + \mathsf{B}(t) \leq (m-b) \cdot C_{max} + b \cdot C_{max} = m \cdot C_{max}$. □

Lemma 5.11. *If $[t_1,t_2]$ is a busy interval for Ψ in $\mathcal{S}$, then $\mathsf{LAG}(\Psi,t_1,\mathcal{S}) + \mathsf{B}(t_1) \geq \mathsf{LAG}(\Psi,t_2,\mathcal{S}) + \mathsf{B}(t_2)$.*

Proof. Since $t_1, t_2 \in [t_1,t_2]$ are busy instants for Ψ, $\mathcal{B}(t_1) = \mathcal{B}(t_2) = \emptyset$ and therefore $\mathsf{B}(t_1) = \mathsf{B}(t_2) = 0$. Also, by Lemma 5.2, $\mathsf{LAG}(\Psi,t_1,\mathcal{S}) \geq \mathsf{LAG}(\Psi,t_2,\mathcal{S})$. Thus, $\mathsf{LAG}(\Psi,t_1,\mathcal{S}) + \mathsf{B}(t_1) \geq \mathsf{LAG}(\Psi,t_2,\mathcal{S}) + \mathsf{B}(t_2)$. □

Lemma 5.12. *If $[t_1,t_2]$ is a blocking non-busy interval for Ψ in $\mathcal{S}$, then any blocking job (i.e., any job that is not in Ψ but executing at some time instant t in $[t_1,t_2]$), must execute continuously in $[t_1^-,t]$.*

Proof. Because $[t_1,t_2]$ is a blocking non-busy interval for Ψ, at any time instant within $[t_1,t_2]$, there is at least one job in Ψ that is pending but not executing. Also, since any job in Ψ has an earlier deadline, or higher priority, than any job not in Ψ, no job that is not in Ψ and not executing at t_1^- can execute in $[t_1,t_2]$. Thus, the lemma follows. □

Lemma 5.13. *If $[t_1,t_2]$ is a blocking non-busy interval for Ψ in $\mathcal{S}$, then $\mathsf{LAG}(\Psi,t_1,\mathcal{S}) + \mathsf{B}(t_1) \geq \mathsf{LAG}(\Psi,t_2,\mathcal{S}) + \mathsf{B}(t_2)$.*

Proof. Let $[t,t']$ be a subinterval in $[t_1,t_2]$ such that $|\mathcal{B}(t)| = |\mathcal{B}(t')|$. By Lemma 5.12, the blocking jobs at every time instant in $[t,t']$ are exactly the jobs in $\mathcal{B}(t)$. Let $\mathcal{P}$ denote the set of processors on which those blocking jobs execute in $[t,t']$. Then,

$$\mathsf{B}(t') = \mathsf{B}(t) - \sum_{s_i \in \mathcal{P}} s_i \cdot (t' - t).$$

Since $[t,t'] \subseteq [t_1,t_2]$ is a blocking non-busy interval, the processors not in $\mathcal{P}$ must execute jobs in Ψ; otherwise, it would be a non-blocking non-busy interval. Therefore,

$$\begin{aligned}
&\mathsf{LAG}(\Psi,t',\mathcal{S}) \\
&= \mathsf{LAG}(\Psi,t,\mathcal{S}) + \mathsf{A}(\mathcal{I},\Psi,t,t') - \mathsf{A}(\mathcal{S},\Psi,t,t') \\
&\leq \mathsf{LAG}(\Psi,t,\mathcal{S}) + U_\tau \cdot (t'-t) - \sum_{s_i \notin \mathcal{P}} s_i \cdot (t'-t).
\end{aligned}$$

Thus,

$$\begin{aligned}
&\mathsf{LAG}(\Psi,t',\mathcal{S}) + \mathsf{B}(t') \\
&\leq \mathsf{LAG}(\Psi,t,\mathcal{S}) + U_\tau \cdot (t'-t) - \sum_{s_i \notin \mathcal{P}} s_i \cdot (t'-t) + \mathsf{B}(t) - \sum_{s_i \in \mathcal{P}} s_i \cdot (t'-t) \\
&= \mathsf{LAG}(\Psi,t,\mathcal{S}) + \mathsf{B}(t) + U_\tau \cdot (t'-t) - (\sum_{s_i \notin \mathcal{P}} s_i + \sum_{s_i \in \mathcal{P}} s_i) \cdot (t'-t) \\
&= \mathsf{LAG}(\Psi,t,\mathcal{S}) + \mathsf{B}(t) + U_\tau \cdot (t'-t) - S_m \cdot (t'-t) \\
&\leq \{\text{since } U_\tau \leq S_m\} \\
&\quad \mathsf{LAG}(\Psi,t,\mathcal{S}) + \mathsf{B}(t).
\end{aligned}$$

That is, for every such subinterval $[t,t'] \subseteq [t_1,t_2]$, we have

$$\mathsf{LAG}(\Psi,t,\mathcal{S}) + \mathsf{B}(t) \geq \mathsf{LAG}(\Psi,t',\mathcal{S}) + \mathsf{B}(t').$$

By induction, the lemma follows. □

Lemma 5.14. *In $\mathcal{S}$, the competing work for $\tau_{k,l}$ plus the blocking work at t_d is at most $L_\tau + m \cdot C_{max}$.*

Proof. Similarly to Lemma 5.5, the competing work pending at t_d is at most $L_\tau + \mathsf{LAG}(\Psi,\mathcal{S},t_d)$. Also, the blocking work at t_d is $\mathsf{B}(t_d)$, so the competing work for $\tau_{i,j}$ plus the blocking work at t_d is at most $L_\tau + \mathsf{LAG}(\Psi,\mathcal{S},t_d) + \mathsf{B}(t_d)$.

Let t' be the latest non-blocking non-busy instant at or before t_d (or time 0 if no such non-blocking non-busy instant exists). Then by Lemma 5.10,

$$\mathsf{LAG}(\Psi,\mathcal{S},t') + \mathsf{B}(t') \leq m \cdot C_{max}.$$

Moreover, by the definition of t', $[t'^{+}, t_d]$ consists of busy intervals and/or blocking non-busy intervals. Therefore, by Lemmas 5.11 and 5.13,

$$\mathsf{LAG}(\Psi, \mathcal{S}, t') + \mathsf{B}(t') \geq \mathsf{LAG}(\Psi, \mathcal{S}, t_d) + \mathsf{B}(t_d).$$

Thus, $\mathsf{LAG}(\Psi, \mathcal{S}, t_d) + \mathsf{B}(t_d) \leq m \cdot C_{max}$ and the lemma follows. □

Lemma 5.15. *Let W be the competing work plus the blocking work for $\tau_{k,l}$ at t_d. Then the job of interest, $\tau_{k,l}$, will complete execution no later than time*

$$t_d + \frac{W - C_k}{S_m} + \frac{C_k}{s_m}.$$

Proof. The proof of this lemma is exactly the same as Lemma 5.6. □

Theorem 5.3. *The response time of an arbitrary job $\tau_{k,l}$ in τ under non-preemptive G-EDF scheduling on π is at most*

$$D_k + \frac{L_\tau + m \cdot C_{max} - C_k}{S_m} + \frac{C_k}{s_m}.$$

Proof. Follows from Lemmas 5.14 and 5.15. □

5.4.2 Improved Bounds

We now show that the response-time bound for non-preemptive G-EDF can also be improved. However, in contrast to the situation in Section 5.3.2, we still have to pessimistically assume the job of interest executes entirely at the minimum speed s_m, since in non-preemptive G-EDF, a scheduled job cannot migrate among processors. Nevertheless, we can still consider the execution of $\tau_{k,l}$ before t_d.

The following three lemmas are similar to Lemmas 5.4, 5.14, and 5.15.

Lemma 5.16. *Under non-preemptive G-EDF, once $\tau_{k,l}$ executes, it will continuously execute until it completes.*

Proof. Follows from the non-preemptive property. □

Lemma 5.17. *For any time instant t at or before t_d, $\mathsf{LAG}(\Psi, \mathcal{S}, t) + \mathsf{B}(t) \leq m \cdot C_{max}$.*

Proof. This proof is exactly the same as that for upper bounding $\mathsf{LAG}(\Psi, \mathcal{S}, t_d) + \mathsf{B}(t_d)$ in Lemma 5.14. □

Lemma 5.18. *Let W be the competing work plus the blocking work for $\tau_{k,l}$ at $r_{k,l}$. Then the job of interest, $\tau_{k,l}$, will complete execution no later than time*

$$r_{k,l} + \frac{W - C_k}{S_m} + \frac{C_k}{s_m}.$$

Proof. By Lemma 5.16, this proof is exactly the same as Lemma 5.15. □

We now upper bound the competing work plus the blocking work for $\tau_{k,l}$ at $r_{k,l}$ by the following lemma.

Lemma 5.19. *The competing work plus the blocking work for $\tau_{k,l}$ at $r_{k,l}$ is at most*

$$U_\tau \cdot D_k + L_\tau + m \cdot C_{max}.$$

Proof. By Definitions 5.3, 5.4, 5.5, and 5.11, the competing work plus the blocking work for $\tau_{k,l}$ at $r_{k,l}$ is at most

$$\begin{aligned}
&L_\tau + \mathsf{A}(\mathcal{I}, \Psi, 0, t_d) - \mathsf{A}(\mathcal{S}, \Psi, 0, r_{k,l}) + \mathsf{B}(r_{k,l}) \\
=\ &\{\text{by Definition 5.2}\} \\
&L_\tau + \mathsf{A}(\mathcal{I}, \Psi, 0, r_{k,l}) + \mathsf{A}(\mathcal{I}, \Psi, r_{k,l}, t_d) - \mathsf{A}(\mathcal{S}, \Psi, 0, r_{k,l}) + \mathsf{B}(r_{k,l}) \\
=\ &\{\text{by (5.5) and (5.6)}\} \\
&L_\tau + \mathsf{LAG}(\Psi, \mathcal{S}, r_{k,l}) + \mathsf{A}(\mathcal{I}, \Psi, r_{k,l}, t_d) + \mathsf{B}(r_{k,l}) \\
\leq\ &\{\text{by (5.4)}\} \\
&U_\tau \cdot (t_d - r_{k,l}) + L_\tau + \mathsf{LAG}(\Psi, \mathcal{S}, r_{k,l}) + \mathsf{B}(r_{k,l}) \\
\leq\ &\{\text{since } t_d = d_{k,l} = r_{k,l} + D_k \text{ and by Lemma 5.17}\} \\
&U_\tau \cdot D_k + L_\tau + m \cdot C_{max}.
\end{aligned}$$

The lemma follows. □

Theorem 5.4. *The response time of an arbitrary job $\tau_{k,l}$ in τ under non-preemptive G-EDF scheduling on π is at most*

$$\frac{U_\tau}{S_m} \cdot D_k + \frac{L_\tau + m \cdot C_{max} - C_k}{S_m} + \frac{C_k}{s_m}.$$

Proof. Follows from Lemmas 5.18 and 5.19. □

5.5 Evaluation

We evaluated the proposed algorithms and derived response-time bounds by randomly generating task sets and then calculating response-time bounds for each task on certain selected platforms.

In the case of identical multiprocessors, the considered platform is implicitly determined by the number of processors. However, for uniform multiprocessors, even if given the number of processors, there are an infinite number of speed combinations to consider. Therefore, there is no way to systematically choose platforms to evaluate by varying the number of processors. Thus, we chose the following four multiprocessors as representatives of different uniform multiprocessors in terms of both processor number and speed combination: $\pi_1 = \{2, 2, 2, 2, 1, 1, 1, 1\}$, $\pi_2 = \{3, 3, 2, 2, 1, 1\}$, $\pi_3 = \{3, 3, 1.5, 1.5, 1.5, 1.5\}$, and $\pi_4 = \{4, 4, 2, 2\}$. We could also normalize the slowest processor of the latter two platforms to be 1.0; however, we chose instead to scale all four platforms to have the same total processor capacity to enable comparisons among different platforms.

In our experiments, we assumed implicit deadlines for simplicity, *i.e.*, relative deadlines are equal to periods ($D_i = T_i$). For a given total utilization cap, we generated a task set by first randomly selecting its task count uniformly over $[1, 20]$. Note that we allow the number of tasks to be less than the number of processors. Next, we randomly assigned a *relative utilization* or *weight*, by uniformly generating a number in $(0, 1]$, for each task. We then scaled the relative utilizations to obtain real utilizations by letting the total utilization match the pre-set cap. Note that, by the scaling step, we may generate tasks with a utilization greater than the fastest processor's speed; that is allowed in our task model and analysis. Since all of the four considered platforms have a total capacity of 12, we varied task-set utilization caps in $[0, 12]$ by increments of 0.2. For each given total utilization cap, we generated 10,000 task sets. The period for each task was selected uniformly within $[10\,\text{ms}, 100\,\text{ms}]$; its worst-case execution requirement was then determined based on its utilization and period.

Results. We evaluated response-time bounds in terms of both absolute values and relative values. The former are directly computed by Theorems 2 and 4; the latter are defined by the ratio of a task's absolute response-time bound and its period. We focus here on the maximum response-time bound for each task set. Figure 5.1 shows absolute response-time bound results while Figure 5.2 shows the relative response-time

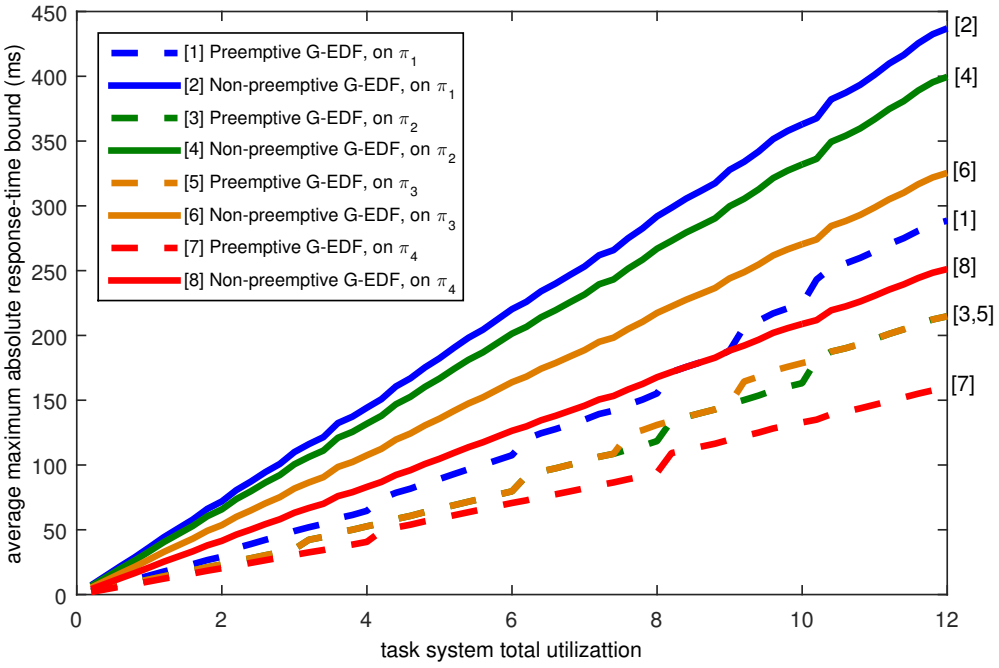


Figure 5.1: Average maximum absolute response-time bounds.

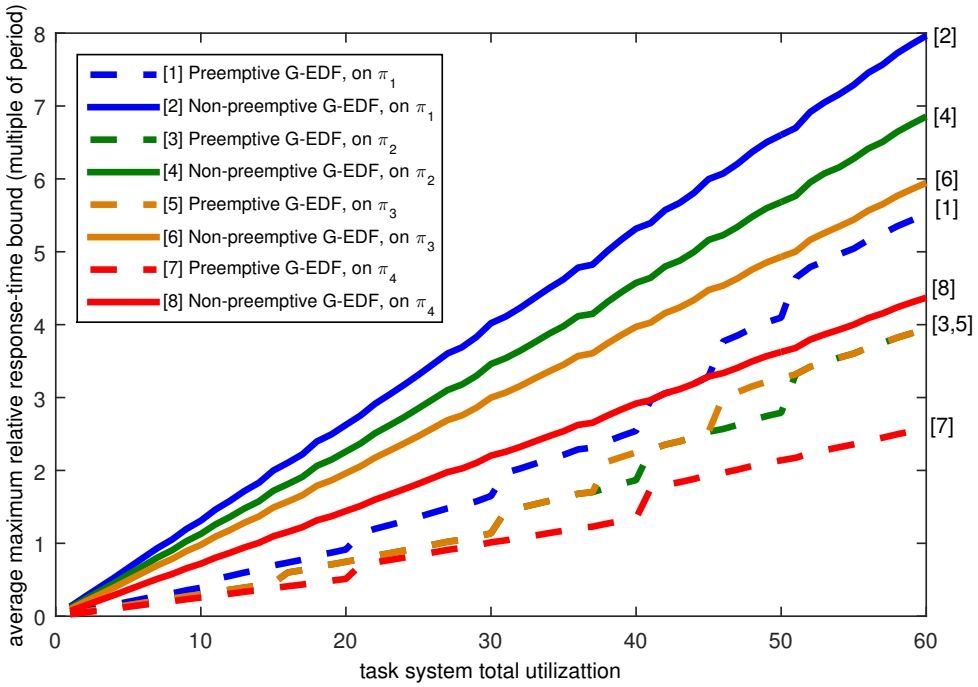


Figure 5.2: Average maximum relative response-time bounds.

bound results. Each data point in each figure is the average of the maximum response-time bounds (absolute or relative) of the 10,000 generated task sets for a given total utilization. The resulting absolute and relative response-time bounds are under 450 ms and 8 periods, respectively. Generally, the lower the total utilization cap, the better the bounds; given a total processor capacity, having fewer processors with faster speeds yields better bounds.

5.6 Chapter Summary

In this chapter, we have considered both preemptive and non-preemptive G-EDF scheduling on uniform multiprocessors in the absence of intra-task precedence constraints. Both ensure bounded job response times for npc-sporadic tasks as long as the underlying multiprocessor platform is not overutilized.

It follows from our work that, on such platforms, different feasibility conditions apply for HRT and SRT npc-sporadic systems. This stands in contrast to the conventional sporadic task model. For the npc-sporadic model, the HRT-feasibility condition is the same as that for the conventional sporadic task model; however, the SRT-feasibility condition for the npc-sporadic model merely requires that the system is not overutilized, in contrast to the more complicated condition for the sporadic case that requires (5.2). Note that both preemptive and non-preemptive G-EDF are SRT-optimal for scheduling npc-sporadic task systems.

Preemptive G-EDF is more greedy in executing jobs on faster processors and hence has a better response-time bound, at the expense of potentially greater preemption and migration frequencies. On the other hand, non-preemptive G-EDF does not preempt or migrate jobs, but its guaranteed response-time bounds are relatively higher. Our analysis for non-preemptive G-EDF applies even under the scheduling rule that, when multiple processors are available to a job, the slowest one is chosen to execute that job. This may yield benefits from an energy point of view.

CHAPTER 6: DAG-BASED TASK SYSTEMS ON UNRELATED HETEROGENEOUS PLATFORMS[1]

The multicore revolution is currently undergoing a second wave of innovation in the form of heterogeneous hardware platforms. In the domain of real-time embedded systems, such platforms may be desirable to use for a variety of reasons. For example, ARM's big.LITTLE multicore architecture (ARM, 2018) enables performance and energy concerns to be balanced by providing a mix of relatively slower, low-power cores and faster, high-power ones. Unfortunately, the move towards greater heterogeneity is further complicating software design processes that were already being challenged on account of the significant parallelism that exists in "conventional" multicore platforms with identical processors. Such complications are impeding advancements in the embedded computing industry today.

Problem considered herein. In this chapter, we report on our efforts towards solving a particular real-time analysis problem concerning heterogeneity motivated by an industrial collaboration. This problem pertains to the processing done by cellular base stations in wireless networks. We refrain from delving into specifics regarding this particular application domain, opting instead for a more abstract treatment of the problem at hand.

This problem involves the scheduling of real-time dataflows on heterogeneous computational elements (CEs), such as CPUs, digital signal processors (DSPs), or one of many types of hardware accelerators. Each dataflow is represented by a directed acyclic graph (DAG), the nodes (resp., edges) of which represent tasks (resp., producer/consumer relationships). A given task is restricted to run on a specific CE type. Task preemption may be impossible for some CEs and should in any case be discouraged. Each DAG has a single source task that is invoked periodically. Intra-task parallelism is allowed in the sense that consecutive jobs (*i.e.*, invocations) of the same task can execute in parallel (but each job executes sequentially). In fact, a later job can finish earlier due to variations in running times.[2] The DAGs to be supported are defined using

[1]Contents of this chapter previously appeared in preliminary form in the following paper:

Yang, K., Yang, M., and Anderson, J. (2016). Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 349–358.

[2]In the considered application domain, the data produced by subsequent jobs can be buffered until all prior jobs of the same task have completed.

a relatively small number of "templates," *i.e.*, many DAGs may exist that are structurally identical. The challenge is to devise a multi-resource, real-time scheduler for supporting dataflows as described here with accompanying per-dataflow end-to-end response-time analysis. That is, an upper bound on the response time of a single invocation of each DAG in a system comprised of such DAGs is required.

In this chapter, we formalize the problem described above and then address it by proposing a scheduling approach and associated end-to-end response-time analysis. In the first part of the chapter, we attack the problem by presenting a transformation process whereby successive task models are introduced such that: **(i)** the first task model directly formalizes the problem above; **(ii)** prior analysis can be applied to the last model to obtain response-time bounds under earliest-deadline-first (EDF) scheduling; and **(iii)** each successive model is a refinement of the prior one in the sense that all DAG-based precedence constraints are preserved. Such a transformation approach was previously used by Liu and Anderson (2010) in work on DAG-based systems, but that work focused on identical multiprocessors. Moreover, our work differs from theirs in that we allow intra-task parallelism. This enables much smaller end-to-end response-time bounds to be derived.

After presenting this transformation process, we discuss two techniques that can reduce the response-time bounds enabled by this process. The first technique exploits the fact that some leeway exists in setting tasks' relative deadlines. By setting more aggressive deadlines for tasks along "long" paths in a DAG, the overall end-to-end response-time bound of that DAG can be reduced. We show that such deadline adjustments can be made by solving a linear program.

The second technique exploits the fact that, in the considered context, DAGs are defined using relatively few templates and typically have quite low utilizations. These facts enable us to reduce response-time bounds by combining many DAGs into one of larger utilization. As a very simple example, two DAGs with a period of 10 time units might be combined into one with a period of 5 time units. A response-time-bound reduction is enabled because these bounds tend to be proportional to periods. In the considered application domain, the extent of combining can be much more extensive: upwards of 40 DAGs may be combinable.

We evaluate our proposed techniques via case-study and schedulability experiments. These experiments show that our techniques can significantly reduce response-time bounds. Furthermore, our analysis supports "early releasing" (Devi, 2006) (see Section 6.6) to improve observed end-to-end response times. We experimentally demonstrate the efficacy of this as well.

Organization. In the following sections, we formalize the considered problem (Section 6.1), present the refinements mentioned above that enable the use of prior analysis (Sections 6.2 and 6.3), show that the

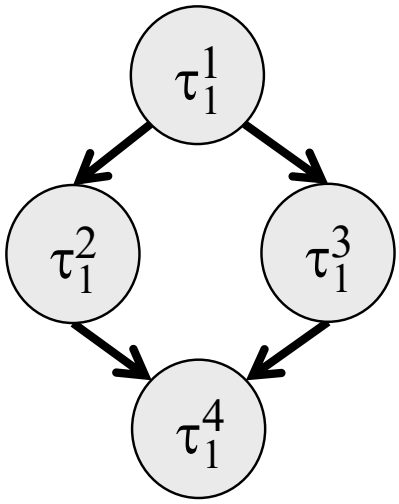


Figure 6.1: A DAG G_1.

bounds arising from this analysis can be improved via linear programming (Section 6.4) and DAG combining (Section 6.5), discuss early releasing (Section 6.6), and present our case-study (Section 6.7) and schedulability (Section 6.8) experiments.

6.1 System Model

In this section, we formalize the dataflow-scheduling problem described earlier and introduce relevant terminology. Each dataflow is represented by a DAG, as discussed earlier.

We specifically consider a system $G = \{G_1, G_2, \ldots, G_N\}$ comprised of N DAGs. The DAG G_i consists of n_i nodes, which correspond to n_i tasks, denoted $\tau_i^1, \tau_i^2, \ldots, \tau_i^{n_i}$. Each task τ_i^v releases a (potentially infinite) sequence of *jobs* $\tau_{i,1}^v$, $\tau_{i,2}^v$, ... The edges in G_i reflect producer/consumer relationships. A particular task τ_i^v's *producers* are those tasks with outgoing edges directed to τ_i^v, and its *consumers* are those with incoming edges directed from τ_i^v. The j^{th} job of task τ_i^v, $\tau_{i,j}^v$, cannot commence execution until the j^{th} jobs of all of its producers have completed; this ensures that its necessary input data is available. Such job dependencies only exist with respect to the *same invocation* of a DAG, and not across different invocations. That is, while jobs must execute sequentially, intra-task parallelism is allowed.

Example 6.1. Figure 6.1 shows an example DAG, G_1. Task τ_1^4's producers are tasks τ_1^2 and τ_1^3, thus for any j, $\tau_{1,j}^4$ needs input data from each of $\tau_{1,j}^2$ and $\tau_{1,j}^3$, so it must wait until those jobs complete. Because intra-task parallelism is allowed, $\tau_{1,j}^4$ and $\tau_{1,j+1}^4$ could potentially execute in parallel. ◊

To simplify analysis, we assume that each DAG G_i has exactly one *source task* τ_i^1, which has only outgoing edges, and one *sink task* $\tau_i^{n_i}$, which has only incoming edges. Multi-source/multi-sink DAGs can be supported with the addition of singular "virtual" sources and sinks that connect multiple sources and sinks, respectively. Virtual sources and sinks have a worst-case execution time (WCET) of zero.

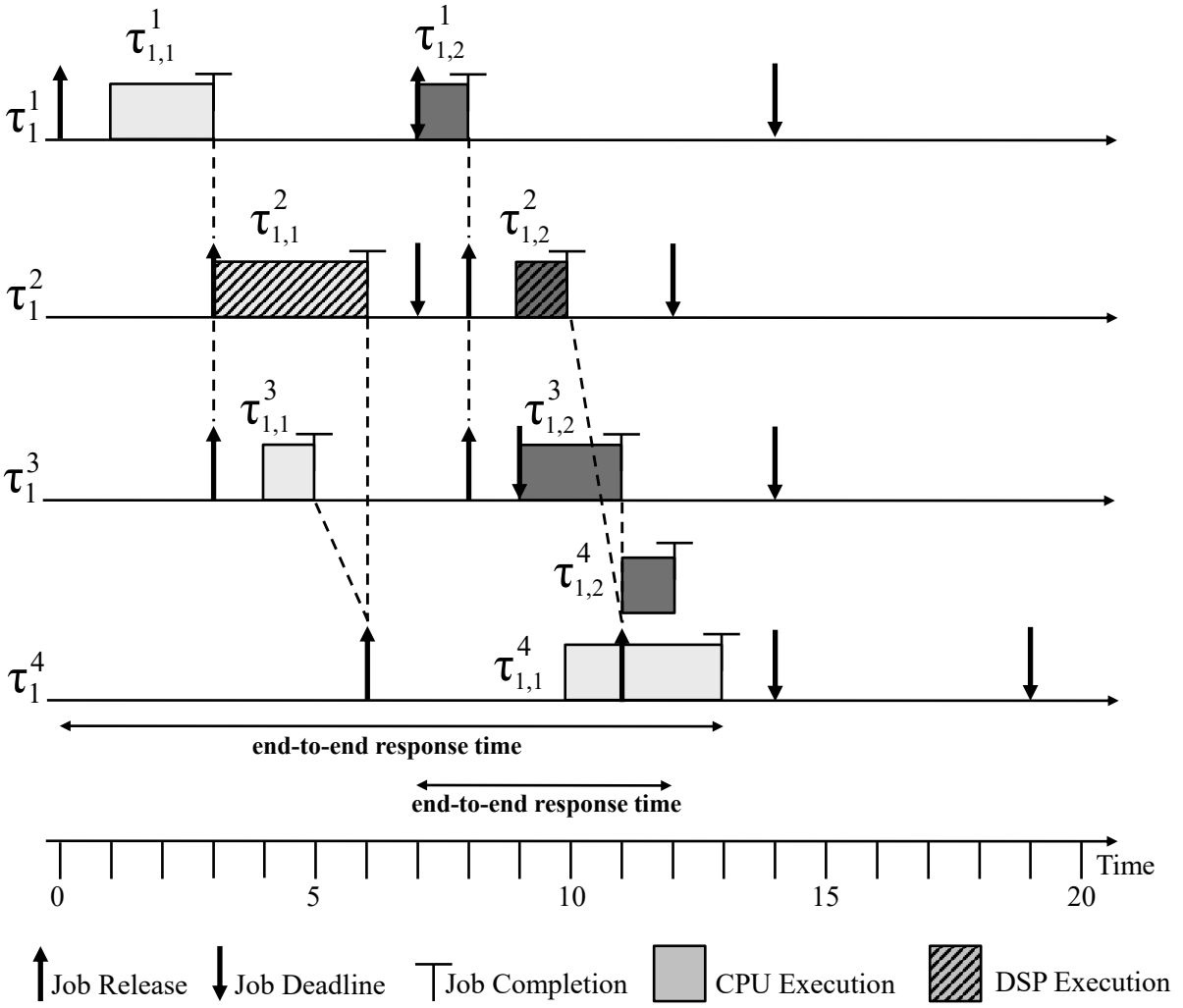
We consider the scheduling of DAGs as just described on a heterogeneous hardware platform consisting of different types of CEs. A given CE might be a CPU, DSP, or some specialized hardware accelerator (HAC). The CEs are organized in M CE *pools*, where each CE pool π_k consists of m_k *identical* CEs. Each task τ_i^v has a parameter P_i^v that denotes the particular CE pool on which it must run, *i.e.*, $P_i^v = \pi_k$ means that each job of τ_i^v must be scheduled on a CE in the CE pool π_k. The WCET of task τ_i^v is denoted C_i^v.

Although the problem description at the beginning of this chapter indicated that source tasks are released periodically, we generalize this to allow sporadic releases, *i.e.*, for the DAG G_i, the job releases of τ_i^1 have a minimum separation time, denoted T_i. A non-source task τ_i^v $(v > 1)$ releases its j^{th} job $\tau_{i,j}^v$ when the j^{th} jobs of all its producer tasks in G_i have completed. That is, letting $r_{i,j}^v$ and $f_{i,j}^v$ denote the release and finish times of $\tau_{i,j}^v$, respectively,

$$r_{i,j}^v = \max\{f_{i,j}^w \mid \tau_i^w \text{ is a producer of } \tau_i^v\}. \tag{6.1}$$

The *response time* of job $\tau_{i,j}^v$ is defined as $f_{i,j}^v - r_{i,j}^v$, and the *end-to-end response time* of the DAG G_i as $f_{i,j}^{n_i} - r_{i,j}^1$.

Example 6.2. Figure 6.2 depicts an example schedule for the DAG G_1 in Figure 6.1, assuming task τ_1^2 is required to execute on a DSP and the other tasks are required to execute on a CPU. The first (resp., second) job of each task has a lighter (resp., darker) shading to make them easier to distinguish. Tasks τ_1^2 and τ_1^3 have only one producer, τ_1^1, so when task τ_1^1 finishes a job at times 3 and 8, τ_1^2 and τ_1^3 release a job immediately. In contrast, task τ_1^4 has two producers, τ_1^2 and τ_1^3. Therefore, τ_1^4 cannot release a job at time 5 when τ_1^3 finishes a job, but rather must wait until time 6 when τ_1^2 also finishes a job. Note that consecutive jobs of the same task might execute in parallel (*e.g.*, $\tau_{1,1}^4$ and $\tau_{1,2}^4$ execute in parallel during $[11, 12)$). Furthermore, for a given task, a later-released job (*e.g.*, $\tau_{1,2}^4$) may even finish earlier than an earlier-released one (*e.g.*, $\tau_{1,1}^4$) due to execution-time variations. ◊



(Assume depicted jobs are scheduled alongside other jobs, which are not shown.)

Figure 6.2: Example schedule for the DAG in G_1 in Figure 6.1.

Scheduling. Since many CEs are non-preemptible, we use the non-preemptive G-EDF scheduling algorithm within each CE pool. The *deadline* of job $\tau_{i,j}^v$ is given by

$$d_{i,j}^v = r_{i,j}^v + D_i^v, \tag{6.2}$$

where D_i^v is the *relative deadline* of task τ_i^v. For example, in the example schedule in Figure 6.2, relative deadlines of $D_1^1 = 7$, $D_1^2 = 4$, $D_1^3 = 6$, and $D_1^4 = 8$ are assumed.

In the context of this chapter, deadlines mainly serve the purpose of determining jobs' priorities rather than strict timing constraints for individual jobs. Therefore, deadline misses are acceptable as long as the end-to-end response time of each DAG can be reasonably bounded.

Utilization. We denote the *utilization* of task τ_i^v by

$$u_i^v = \frac{C_i^v}{T_i}. \tag{6.3}$$

We also use Γ_k to denote the set of tasks that are required to execute on the CE pool π_k, *i.e.*,

$$\Gamma_k = \{\tau_i^v \mid P_i^v = \pi_k\}. \tag{6.4}$$

The overutilization of a CE pool could cause unbounded response times, so we require for each k,

$$\sum_{\tau_i^v \in \Gamma_k} u_i^v \leq m_k. \tag{6.5}$$

6.2 Offset-Based Independent Tasks

In this section, we present a second task model, which is a refinement of that just presented, as will be shown in Section 6.3. The prior model is somewhat problematic because of difficult-to-analyze dependencies among jobs. In particular, by (6.1), the release times of jobs of non-source tasks depend on the finish times of other jobs, and hence on their execution times. By (6.2), deadlines (and hence priorities) of jobs are affected by similar dependencies.

In order to ease analysis difficulties associated with such job dependencies, we introduce here the *offset-based independent task (obi-task) model*. Under this model, tasks are partitioned into groups. The i^{th}

such group consists of tasks denoted $\tau_i^1, \tau_i^2, \ldots, \tau_i^{n_i}$, where τ_i^1 is a designated *source* task that releases jobs sporadically with a minimum separation of T_i. That is, for any positive integer j,

$$r_{i,j+1}^1 - r_{i,j}^1 \geq T_i. \tag{6.6}$$

Job releases of each non-source task τ_i^v are governed by a new parameter Φ_i^v, called the *offset* of τ_i^v. Specifically, τ_i^v releases its j^{th} job exactly Φ_i^v time units after the release time of the j^{th} job of the source task τ_i^1 of its group. That is,

$$r_{i,j}^v = r_{i,j}^1 + \Phi_i^v. \tag{6.7}$$

For consistency, we define

$$\Phi_i^1 = 0. \tag{6.8}$$

Under the obi-task model, a job of a task τ_i^v can be scheduled at any time after its release *independently* of the execution of any other jobs, *even jobs of the same task* τ_i^v.

The definitions so far have dealt with job releases. Additionally, the two per-task parameters C_i^v and P_i^v from Section 6.1 are retained with the same definitions.

The following property shows that every obi-task τ_i^v has a minimum job-release separation of T_i.

Property 6.1. *For any obi-task* τ_i^v, $r_{i,j+1}^v - r_{i,j}^v \geq T_i$.

Proof.

$$\begin{aligned} r_{i,j+1}^v - r_{i,j}^v = {} & \{\text{by (6.7)}\} \\ & (r_{i,j+1}^1 + \Phi_i^v) - (r_{i,j}^1 + \Phi_i^v) \\ \geq {} & \{\text{by (6.6)}\} \\ & T_i \end{aligned}$$

□

6.3 Response-Time Bounds

In this section, we establish two results that enable prior work to be leveraged to establish response-time bounds for DAG-based task systems. First, we show that, under the obi-task model with arbitrary

offset settings, per-task response-time bounds can be derived by exploiting prior work pertaining to npc-sporadic tasks, which were summarized in Chapter 5. Second, we show that, by properly setting offsets, any DAG-based task system can be transformed to a corresponding obi-task system.

6.3.1 Response-Time Bounds for Obi-Tasks

An *npc-sporadic task* τ_i is specified by (C_i, T_i, D_i), where C_i is its WCET, T_i is the minimum separation time between consecutive job releases of τ_i, and D_i is its relative deadline. As before, τ_i's utilization is $u_i = C_i/T_i$.

The main difference between the conventional sporadic task model and the npc-sporadic task model is that the former requires successive jobs of each task to execute in sequence while the latter allows them to execute in parallel. That is, under the conventional sporadic task model, job $\tau_{i,j+1}$ cannot commence execution until its predecessor $\tau_{i,j}$ completes, even if $r_{i,j+1}$, the release time of $\tau_{i,j+1}$, has elapsed. In contrast, under the npc-sporadic task model, any job can execute as soon as it is released. Note that, although we allow intra-task parallelism, each individual job still must execute sequentially.

Chapter 5, we investigated the G-EDF scheduling of npc-sporadic tasks on uniform multiprocessor platforms where different processors may have different speeds. By setting each processor's speed to be 1.0, the following theorem follows from Theorem 5.4.

Theorem 6.1. *(Follows from Theorem 5.4) Consider the scheduling of a set of npc-sporadic tasks τ on m identical multiprocessors. Under non-preemptive G-EDF, each npc-task $\tau_i \in \tau$ has the following response-time bound, provided $\sum_{\tau_l \in \tau} u_l \leq m$.*

$$\frac{1}{m}\left(D_i \cdot \sum_{\tau_l \in \tau} u_l + \sum_{\tau_l \in \tau}\left(u_l \cdot \max_{\tau_l \in \tau}\{0, T_l - D_l\}\right)\right) + \max_{\tau_l \in \tau}\{C_l\} + \frac{m-1}{m}C_i.$$

We now show that Theorem 6.1 can be applied to obtain per-task response-time bounds for any obi-task set.

Concrete vs. non-concrete. A *concrete* sequence of job releases that satisfies a task's specification (under either the obi- or npc-sporadic task model) is called an *instantiation* of that task. An instantiation of a task *set* is defined similarly. In contrast, a task or a task set that can have multiple (potentially infinite) instantiations satisfying its specification (*e.g.*, minimum release separation) is called *non-concrete*.

By Property 6.1, *any* instantiation of an obi-task τ_i^v is an instantiation of the npc-sporadic task $\tau_i^v = (C_i^v, T_i, D_i^v)$. Hence, any instantiation of an obi-task set $\{\tau_i^v \mid P_i^v = \pi_k\}$ is an instantiation of the npc-sporadic task set $\{\tau_i^v \mid P_i^v = \pi_k\}$. Also, since obi-tasks execute independently of one another, obi-tasks executing in different CE pools cannot affect each other. Since each CE pool π_k has m_k identical processors, the problem we must consider is that of scheduling an instantiation of the npc-sporadic task set $\{\tau_i^v \mid P_i^v = \pi_k\}$ on m_k identical processors. Since Theorem 6.1 applies to a non-concrete npc-sporadic task set, it applies to every concrete instantiation of such a task set. Thus, we have the following response-time bound for each obi-task τ_i^v:

$$R_i^v = \frac{1}{m_k}\left(D_i^v \cdot \sum_{\tau_l^w \in \Gamma_k} u_l^w + \sum_{\tau_l^w \in \Gamma_k} (u_l^w \cdot \max\{0, T_l - D_l^w\})\right) + \max_{\tau_l^w \in \Gamma_k}\{C_l^w\} + \frac{m_k - 1}{m_k} C_i^v, \tag{6.9}$$

where $\Gamma_k = \{\tau_l^w \mid P_l^w = \pi_k\}$.

Note that (6.9) is applicable as long as all relative deadlines are non-negative, and applies assuming *any arbitrary* offset setting.

6.3.2 From DAG-Based Task Sets to Obi-Task Sets

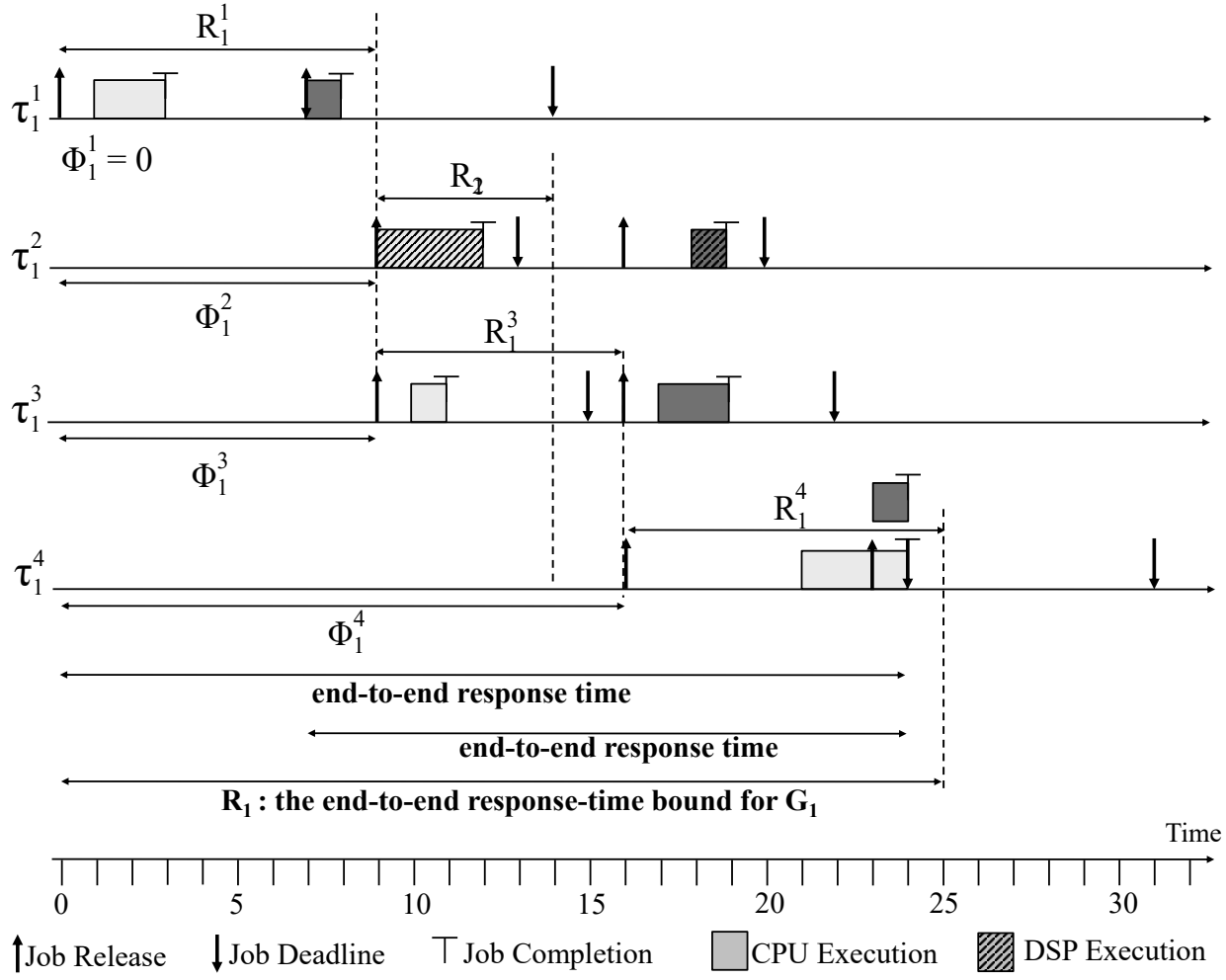
We now show that, by properly setting offsets, any DAG-based task set can be transformed to an obi-task set, and per-DAG end-to-end response-time bounds can be derived by leveraging the obi-task response-time bounds just stated.

Any DAG-based task set can be implemented by an obi-task set in an obvious way: each DAG becomes an obi-task group with the same task designated as its source, and all T_i, C_i^v, and P_i^v parameters are retained without modification. What is less obvious is how to define task offsets under the obi-task model. This is done by setting each Φ_i^v ($v \neq 1$) parameter to be a *constant* such that

$$\Phi_i^v \geq \max_{\tau_i^k \in prod(\tau_i^v)} \{\Phi_i^k + R_i^k\}, \tag{6.10}$$

where $prod(\tau_i^v)$ denotes the set of obi-tasks corresponding to the DAG-based tasks that are the producers of the DAG-based task τ_i^v in G_i, and R_i^k denotes a *response-time bound* for the obi-task τ_i^k. For now, we assume that R_i^k is known, but later, we will show how to compute it.

Example 6.3. Consider again the DAG G_1 in Figure 6.1. Assume that, after applying the above transformation, the obi-tasks have response-time bounds of $R_1^1 = 9$, $R_1^2 = 5$, $R_1^3 = 7$, and $R_1^4 = 9$, respectively. Then,



(Assume depicted jobs are scheduled alongside other jobs, which are not shown.)

Figure 6.3: Example schedule of the obi-tasks corresponding to the DAG-based tasks in G_1 in Figure 6.1.

we can set $\Phi_1^1 = 0$, $\Phi_1^2 = 9$, $\Phi_1^3 = 9$, and $\Phi_1^4 = 16$, respectively, and satisfy (6.10). With these response-time bounds, the end-to-end response-time bound that can be guaranteed is determined by R_1^1, R_1^3, and R_1^4 and is given by $R_1 = 25$. Figure 6.3 depicts a possible schedule for these obi-tasks and illustrates the transformation. Like in Figure 6.2, the first (resp., second) job of each task has a lighter (resp., darker) shading, and intra-task parallelism is possible (*e.g.*, $\tau_{1,1}^4$ and $\tau_{1,2}^4$ in time interval $[23, 24)$). ◊

The following properties follow from this transformation process. According to Property 6.2, a DAG-based task set can be implemented by a corresponding set of obi-tasks, and all producer/consumer constraints

in the DAG-based specification will be implicitly guaranteed, provided the offsets of the obi-tasks are properly set (*i.e.*, satisfy (6.10)).

Property 6.2. *If τ_i^k is a producer of τ_i^v in the DAG-based task system, then for the j^{th} jobs of the corresponding two obi-tasks, $f_{i,j}^k \leq r_{i,j}^v$.*

Proof. By (6.7), $r_{i,j}^k = r_{i,j}^1 + \Phi_i^k$, and by the definition of R_i^k, $f_{i,j}^k \leq r_{i,j}^k + R_i^k$. Thus,

$$f_{i,j}^k \leq r_{i,j}^1 + \Phi_i^k + R_i^k. \tag{6.11}$$

By (6.7), $r_{i,j}^v = r_{i,j}^1 + \Phi_i^v$, and by (6.10), $\Phi_i^v \geq \Phi_i^k + R_i^k$. Thus,

$$r_{i,j}^v \geq r_{i,j}^1 + \Phi_i^k + R_i^k. \tag{6.12}$$

By (6.11) and (6.12), $f_{i,j}^k \leq r_{i,j}^v$. □

Property 6.3 shows how to compute an end-to-end response-time bound R_i.

Property 6.3. *In the obi-task system, for each j, all jobs $\tau_{i,j}^1, \tau_{i,j}^2, \cdots, \tau_{i,j}^{n_i}$ finish their execution within R_i time units after r_i^1, where*

$$R_i = \Phi_i^{n_i} + R_i^{n_i}. \tag{6.13}$$

Proof. By (6.7) and the definition of R_i^v, $\tau_{i,j}^v$ finishes by time $r_{i,j}^1 + \Phi_i^v + R_i^v$. Thus, $\tau_{i,j}^{n_i}$ in particular finishes within $\Phi_i^{n_i} + R_i^{n_i} = R_i$ time units after r_i^1. Also, by (6.10), $\Phi_i^v + R_i^v \leq \Phi_i^{n_i}$, since $\tau_i^{n_i}$ is the *single* sink in G_i. Because $\Phi_i^{n_i} \leq R_i$, this implies that, for any v, $\tau_{i,j}^v$ finishes within R_i time units after r_i^1. □

Thus, a DAG-based task set can be transformed to an obi-task set with the same per-task parameters. Given these per-task parameters, a response-time bound for each obi-task can be computed by (6.9) for *any* arbitrary offset setting. Then, we can properly set the offsets for each obi-task according to (6.10) by considering the corresponding tasks in each DAG in topological order (Cormen et al., 2001), starting with $\Phi_i^1 = 0$ for each source task τ_i^1, by (6.8). By Property 6.2, the resulting obi-task set satisfies all requirements of the original DAG-based task system, and by Property 6.3, an end-to-end response-time bound R_i can be computed for each DAG G_i.

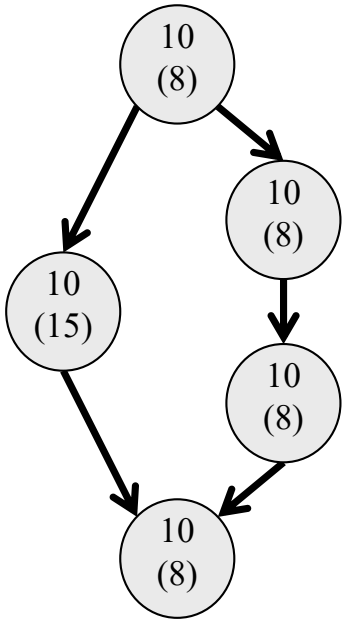


Figure 6.4: More highly prioritizing the right-side path in this DAG decreases its end-to-end response-time bound.

Note that the response-time bound for a virtual source/sink is not computed by (6.9), but is zero by definition, since its WCET is zero. Any job of such a task completes in zero time as soon as it is released.

6.4 Setting Relative Deadlines

In the prior sections, we showed that, by applying our proposed transformation techniques, an end-to-end response-time bound for each DAG can be established, given arbitrary but fixed relative-deadline settings. That is, given $D_i^v \geq 0$ for any i, v, we can compute corresponding end-to-end response-time bounds (*i.e.*, R_i for each i) by (6.9), (6.8), (6.10), and (6.13).

Similar DAG transformation approaches have been presented previously (Elliott et al., 2014; Liu and Anderson, 2010), but under the assumption that intra-task precedence constraints exist (*i.e.*, jobs of the same task must execute in sequence). Moreover, in this prior work, per-task relative deadlines have been defined in a DAG-oblivious way. By considering the actual structure of such a DAG, it may be possible to reduce its end-to-end response-time bound by setting its tasks' relative deadlines so as to favor certain critical paths.

Consider, for example, the DAG illustrated in Figure 6.4. Suppose that the prior analysis yields a response-time bound of 10 for each task, as depicted within each node. The corresponding end-to-end response-time bound would then be 40 and is obtained by considering the right-side path. Now, suppose that we alter the tasks' relative-deadline settings to favor the tasks along this path at the possible expense of the

remaining task on the left. Further, suppose this modification changes the per-task response-time bounds to be as depicted in parentheses. Then, this modification would have the impact of reducing the end-to-end bound to 32.

In this section, we show that the problem of determining the "best" relative-deadline settings can be cast as a linear-programming problem, which can be solved in polynomial time. The proposed linear program (LP) is developed in the next two subsections.

6.4.1 Linear Program

In our LP, there are three variables per task τ_i^v: D_i^v, Φ_i^v, and R_i^v. The parameters T_i, C_i^v, and m_k are viewed as constants. Thus, there are $3|V|$ variables in total, where $|V|$ is the total number of tasks (*i.e.*, nodes) across all DAGs in the system. Before stating the required constraints, we first establish the following theorem, which shows that a relative-deadline setting of $D_x^y > T_x$ is pointless to consider.

Theorem 6.2. *If $D_x^y > T_x$, then by setting $D_x^y = T_x$, R_x^y, the response-time bound of task τ_x^y, will decrease, and each other task's response-time bound will remain the same.*

Proof. To begin, note that, by (6.9), τ_x^y does not impact the response-time bounds of those tasks executing on CE pools other than P_x^y. Therefore, the response-time bounds $\{R_i^v \mid P_i^v \neq P_x^y\}$ for such tasks are not altered by any change to D_x^y.

In the remainder of the proof, we consider a task τ_i^v such that $P_i^v = P_x^y$. Let R_i^v and $R_i'^v$ denote the response-time bounds for τ_i^v before and after, respectively, reducing D_x^y to T_x. If $i = x$ and $v = y$, then by (6.9),

$$\begin{aligned} R_x^y - R_x'^y &= \frac{D_x^y - T_x}{m_k} \cdot \sum_{\tau_l^w \in \tau^k} u_l^w + \frac{u_x^y}{m_k}(\max\{0, T_x - D_x^y\} - \max\{0, T_x - T_x\}) \\ &> \{\text{since } D_x^y > T_x\} \\ &\quad 0. \end{aligned}$$

Alternatively, if $i \neq x$ or $v \neq y$, then by (6.9),

$$\begin{aligned} R_i^v - R_i'^v &= \frac{u_x^y}{m_k}(\max\{0, T_x - D_x^y\} - \max\{0, T_x - T_x\}) \\ &= \{\text{since } D_x^y > T_x\} \\ &\quad 0. \end{aligned}$$

Thus, the theorem follows. □

By Theorem 6.2, the reduction of D_x^y mentioned in the theorem does not increase the response-time bound for any task. By (6.10), this implies that none of the offsets, $\{\Phi_i^v\}$, needs to be increased. Therefore, by Property 6.3, no end-to-end response-time bound increases. These properties motivate our first set of linear constraints.

Constraint Set (i): For each task τ_i^v,

$$0 \leq D_i^v \leq T_i.$$

$2|V|$ individual linear inequalities arise from this constraint set, where $|V|$ is the total number of tasks.

Another issue we must address is that of ensuring that the offset settings, given by (6.10), are encoded in our LP. This gives rise to the next constraint set.

Constraint Set (ii): For each edge from τ_i^w to τ_i^v in a DAG G_i,

$$\Phi_i^v \geq \Phi_i^w + R_i^w.$$

There are $|E|$ distinct constraints in this set, where $|E|$ is the total number of edges in all DAGs in this system.

Finally, we have a set of constraints that are linear equality constraints.

Constraint Set (iii): With Constraints Set (i), it is clear that we can re-write (6.9) as follows, for each task τ_i^v,

$$R_i^v = \frac{1}{m_k}\left(D_i^v \cdot \sum_{\tau_l^w \in \Gamma_k} u_l^w + \sum_{\tau_l^w \in \Gamma_k} \left(u_l^w \cdot (T_l - D_l^w)\right)\right) + \max_{\tau_l^w \in \Gamma_k}\{C_l^w\} + \frac{m_k - 1}{m_k} C_i^v, \tag{6.14}$$

where $\Gamma_k = \{\tau_l^w | P_l^w = \pi_k\}$. Moreover, by (6.8), for each DAG G_i,

$$\Phi_i^1 = 0.$$

Constraint Set (iii) yields $|V| + |G|$ linear equations, where $|G|$ denotes the number of DAGs.

Constraint Sets (i), (ii), and (ii) fully specify our LP, with the exception of the objective function. In this LP, there are $3|V|$ variables, $2|V| + |E|$ inequality constraints, and $|V| + |G|$ linear equality constraints.

6.4.2 Objective Function

Different objective functions can be specified for our LP that optimize end-to-end response-time bounds in different senses. Here, we consider a few examples.

Single-DAG systems. For systems where only a single DAG exists, the optimization criterion is rather clear. In order to optimize the end-to-end response-time bound of the single DAG, the objective function should minimize the end-to-end response-time bound of the only DAG, G_1. That is, the desired LP is as follows.

$$\begin{array}{ll} \text{minimize} & \Phi_1^{n_1} + R_1^{n_1} \\ \text{subject to} & \text{Constraint Sets (i), (ii), and (iii)} \end{array}$$

Multiple-DAG systems. For systems containing multiple DAGs, choices exist as to the optimization criteria to consider. We list three here.

Minimizing the average end-to-end response-time bound:

$$\begin{array}{ll} \text{minimize} & \sum_i (\Phi_i^{n_i} + R_i^{n_i}) \\ \text{subject to} & \text{Constraint Sets (i), (ii), and (iii)} \end{array}$$

Minimizing the maximum end-to-end response-time bound:

$$\begin{array}{ll} \text{minimize} & Y \\ \text{subject to} & \forall i: \quad \Phi_i^{n_i} + R_i^{n_i} \leq Y \\ & \text{Constraint Sets (i), (ii), and (iii)} \end{array}$$

Minimizing the maximum proportional end-to-end response-time bound:

$$\begin{array}{ll} \text{minimize} & Y \\ \text{subject to} & \forall i: \quad (\Phi_i^{n_i} + R_i^{n_i})/T_i \leq Y \\ & \text{Constraint Sets (i), (ii), and (iii)} \end{array}$$

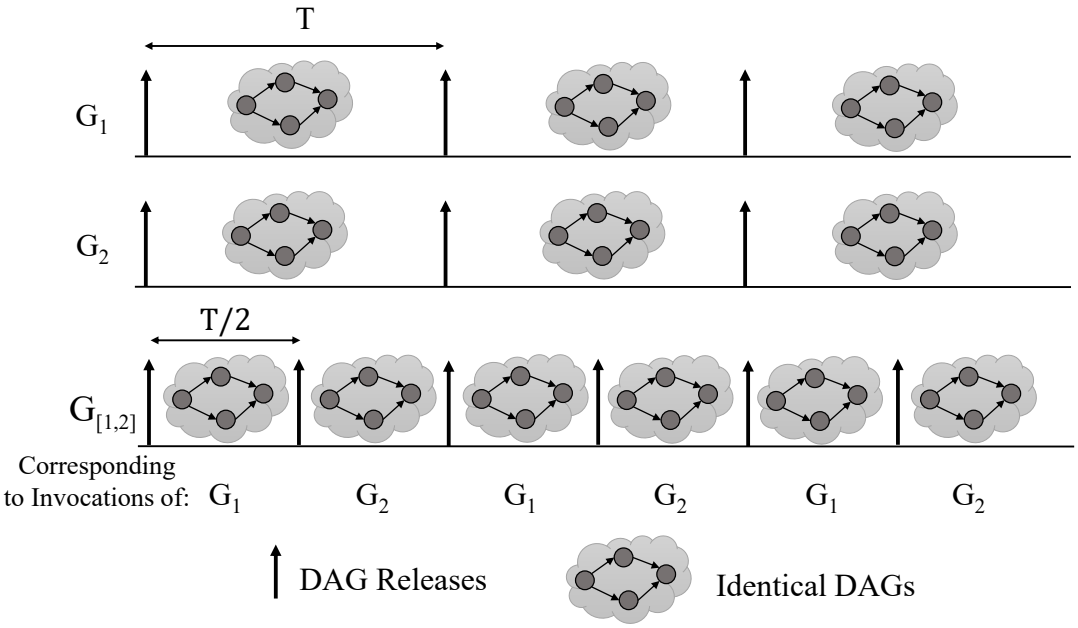


Figure 6.5: Illustration of DAG combining.

6.5 DAG Combining

In the application domain that motivates our work, the DAGs to be scheduled are typically of quite low utilizations and are defined based on a relatively small number of templates that define various computational patterns. Two DAGs defined using the same template are structurally identical: they are defined by graphs that are isomorphic, corresponding nodes from the two graphs perform identical computations, the source nodes are released at the same time, *etc.* Such structurally identical graphs can be combined into one graph with a reduced period and larger utilization, as long any overutilization of the underlying hardware platform is avoided. Such combining can be a very effective technique, because as the experiments presented later show, our response-time bounds tend to be proportional to periods.

We illustrate this idea with a simple example. Consider two DAGs G_1 and G_2 with a common period of T that are structurally identical. A schedule of these two DAGs is illustrated abstractly at the top of Figure 6.5. As illustrated at the bottom of the figure, if these two DAGs are combined, then they are replaced by a structurally identical graph, denoted here as $G_{[1,2]}$, with a period of $T/2$. With this change, the provided response-time bounds have to be slightly adjusted. For example, if $G_{[1,2]}$ has a response-time bound of $R_{[1,2]}$, then this would also be a response-time bound for G_1, but that for G_2 would be $R_{[1,2]} + \frac{T}{2}$, because in combining the two graphs, the releases of G_2 are effectively shifted forward by $\frac{T}{2}$ time units. While this graph

combining idea is really quite simple, the experiments presented later suggest that it can have a profound impact in the considered application domain. In particular, in that domain, per-DAG utilizations are low enough that upwards of 40 DAGs can be combined into one. Thus, the actual period reduction is not merely by a factor of $\frac{1}{2}$ but by a factor as high as $\frac{1}{40}$.

6.6 Early Releasing

Transforming a DAG-based task system to a corresponding obi-task system enabled us to derive an end-to-end response-time *bound* for each DAG. However, such a transformation may actually cause *observed* end-to-end response times at runtime to increase, because the offsets introduced in the transformation may prevent a job from executing even if all of its producers have already finished. For example, in Figure 6.3, $\tau^3_{1,1}$ cannot execute until time 9, even though its corresponding producer job, $\tau^1_{1,1}$, has finished by time 3.

Observed response times can be improved under deadline-based scheduling without altering analytical response-time bounds by using a technique called *early releasing* (Devi, 2006). When early releasing is allowed, a job is eligible for execution as soon as all of its corresponding producer jobs have finished, even if this condition is satisfied before its actual release time.

Early releasing does not affect the response-time analysis for npc-sporadic tasks because that analysis is based on the total demand for processing time due to jobs with deadlines at or before a particular time instant. Early releasing does not change upper bounds on such demand, because every job's actual release time and hence deadline are unaltered by early releasing. Thus, the response-time bounds and therefore the end-to-end response-time bounds previously established without early releasing still hold with early releasing.

Example 6.4. Considering G_1 in Figure 6.1 again, Figure 6.3 is a possible schedule, without early releasing, for the obi-tasks that implement G_1, as discussed earlier. When we allow early releasing, we do not change any release times or deadlines, but simply allow a job to become eligible for execution before its release time provided its producers have finished. Figure 6.6 depicts a possible schedule where early releasing is allowed, assuming the same releases and deadlines as in Figure 6.3. Several jobs (*e.g.*, $\tau^2_{1,1}$, $\tau^2_{1,2}$, $\tau^3_{1,2}$, $\tau^4_{1,1}$, and $\tau^4_{1,2}$) now commence execution before their release times. As a result, observed end-to-end response times are reduced, while still retaining all response-time bounds (per-task and end-to-end). $\Diamond$

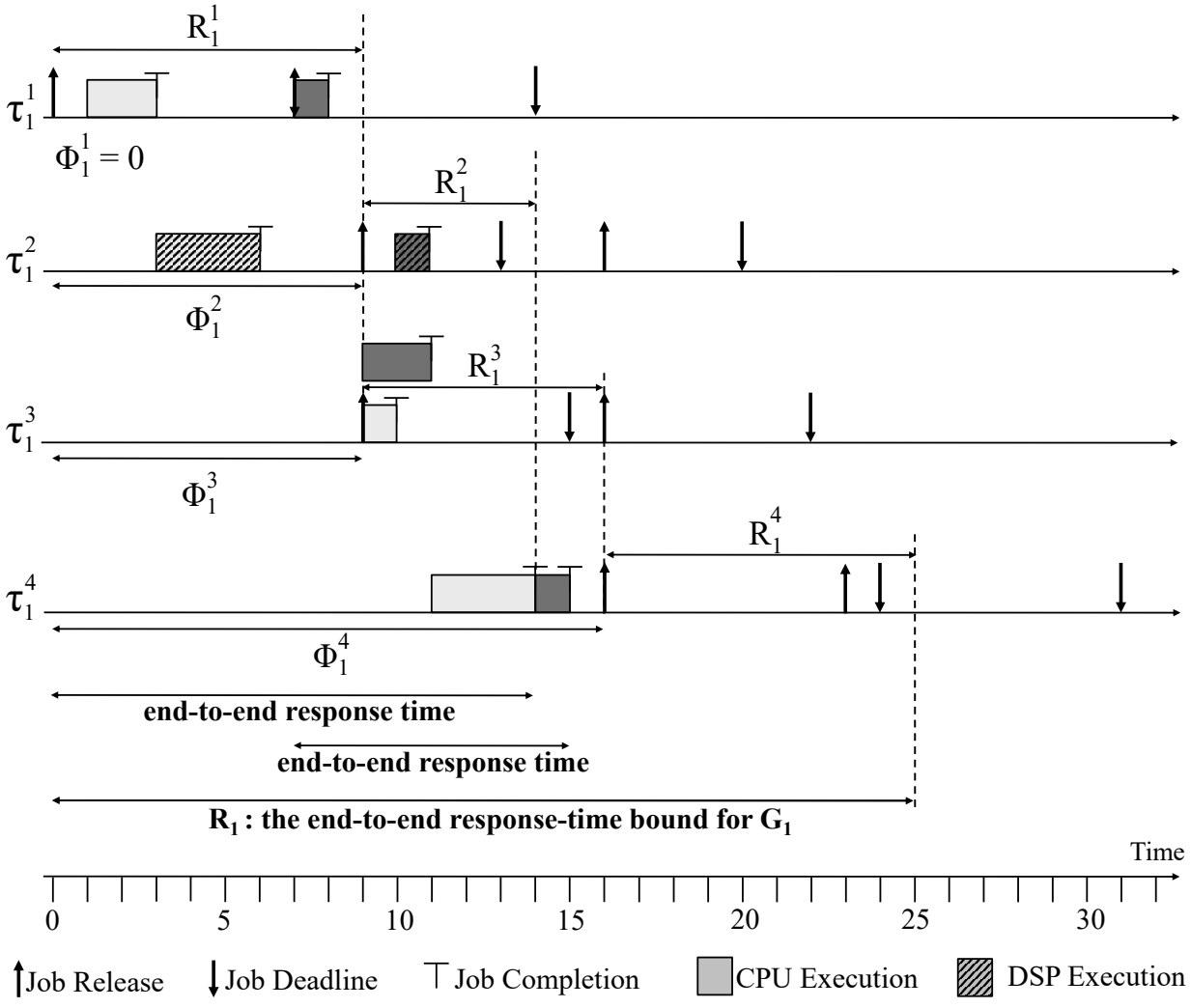


Figure 6.6: Example schedule of the obi-tasks corresponding to the DAG-based tasks in G_1 in Figure 6.1, when early releasing is allowed.

6.7 Case Study

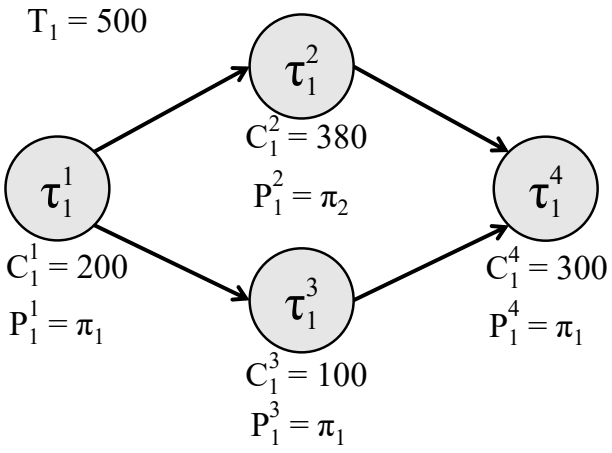
To illustrate the computational details of our analysis, we consider here a case-study system consisting of three DAGs, G_1, G_2, and G_3, which are specified in Figure 6.7. π_1 is a CE pool consisting of two identical CPUs, and π_2 is a CE pool consisting of two identical DSPs. Thus, $m_1 = m_2 = 2$. These three DAGs have fewer nodes and higher utilizations than typically found in our considered application domain. However, one can imagine that these graphs were obtained from combining many identical graphs of lower utilization. While it would have been desirable to consider larger graphs with more nodes, graphs from our chosen domain typically have tens of nodes, and this makes them rather unwieldy to discuss. Still, the general conclusions we draw here are applicable to larger graphs.

Utilization check. First, we must calculate the total utilization of all tasks assigned to each CE pool to make sure that neither is overutilized. We have $\sum_{\tau_i^v \in \Gamma_1} u_i^v = (200+100+300)/500 + (133+78+197+73+5)/1000 = 1.686 < 2$, and $\sum_{\tau_i^v \in \Gamma_2} u_i^v = 380/500 + (16+83+242)/1000 = 1.101 < 2$.

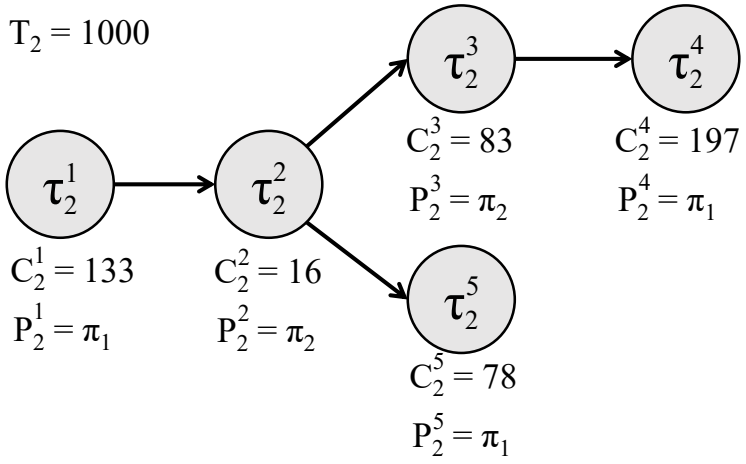
Virtual source/sink. Note that all DAGs have a single source and sink, except for G_2, which has two sinks. For it, we connect its two sinks to a single virtual sink τ_2^6, which has a WCET of 0 and a response-time bound of 0. We call the resulting DAG G_2', which is shown in Figure 6.8.

Implicit deadlines. We now show how to compute response-time bounds assuming implicit deadines, *i.e.*, $D_1^v = 500$ for $1 \leq v \leq 4$, $D_2^v = 1000$ for $1 \leq v \leq 5$ (the relative deadline of the virtual sink is irrelevant), and $D_3^v = 1000$ for $1 \leq v \leq 3$. In order to derive an end-to-end response-time bound, we first transform the original DAG-based tasks into obi-tasks as described in Section 6.2. Next, we calculate a response-time bound R_i^v for each obi-task τ_i^v by (6.9). The resulting task response-time bounds, $\{R_i^v\}$, are listed in Table 6.1. Note that, as a virtual sink, the response-time bound for the virtual sink τ_2^6 does not need to be computed by (6.9), but is 0 by definition. By (6.8) and (6.10), the offsets of the obi-tasks can now be computed in topological order with respect to each DAG. The resulting offsets, $\{\Phi_i^v\}$, are also shown in Table 6.1. Finally, by Property 6.3, we have an end-to-end response-time bound for each DAG: $R_1 = \Phi_1^4 + R_1^4 = 2538.25$, $R_2 = \Phi_2^6 + R_2^6 = 4361.5$, and $R_3 = \Phi_3^3 + R_3^3 = 3376.5$.

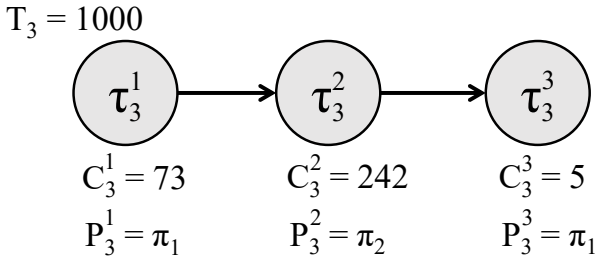
LP-based deadline settings. If we use LP techniques to optimize end-to-end response-time bounds, then choices exist regarding the objective function, because our system has multiple DAGs. We consider three choices here.



(a) G_1



(b) G_2



(c) G_3

Figure 6.7: DAGs in the case-study system. G_2 has two sinks, so to analyze it, a virtual sink τ_2^6 must be added that has a WCET of 0 and a response-time bound of 0. We show the resulting graph in Figure 6.8.

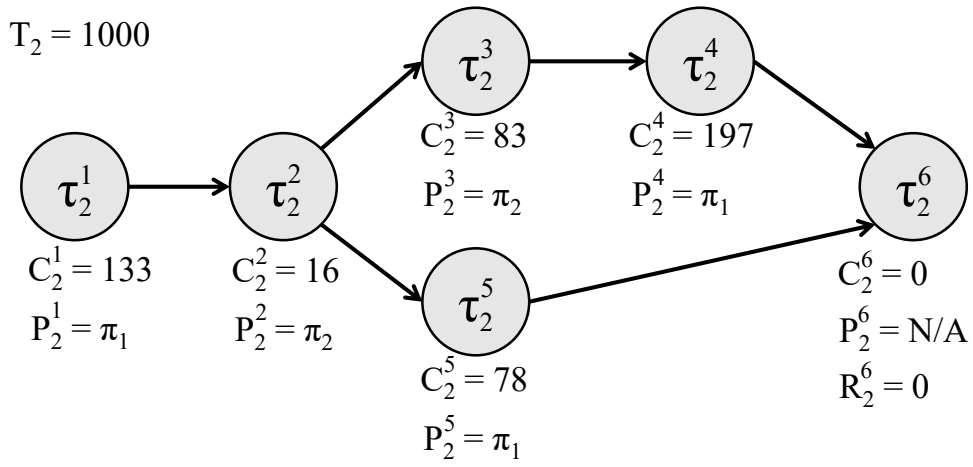


Figure 6.8: G'_2, where a virtual sink is created for G_2.

R_1^1	R_1^2	R_1^3	$\boldsymbol{R_1^4}$	R_2^1	R_2^2	R_2^3	R_2^4	R_2^5	$\boldsymbol{R_2^6}$	R_3^1	R_3^2	$\boldsymbol{R_3^3}$
821.5	845.25	771.5	**871.5**	1209.5	938.5	972	1241.5	1182	**0**	1179.5	1051.5	**1145.5**
Φ_1^1	Φ_1^2	Φ_1^3	$\boldsymbol{\Phi_1^4}$	Φ_2^1	Φ_2^2	Φ_2^3	Φ_2^4	Φ_2^5	$\boldsymbol{\Phi_2^6}$	Φ_3^1	Φ_3^2	$\boldsymbol{\Phi_3^3}$
0	821.5	821.5	**1666.75**	0	1209.5	2148	3120	2148	**4361.5**	0	1179.5	**2231**

Table 6.1: Case-study task response-time bounds and obi-task offsets assuming implicit deadlines. Bold entries denote sinks.

Minimizing the average end-to-end response-time bound. For this choice, relative-deadline settings, obi-task response-time bounds, and obi-task offsets are as shown in Table 6.2 (a). The resulting end-to-end response-time bounds are $R_1 = \Phi_1^4 + R_1^4 = 3134.5$, $R_2 = \Phi_2^6 + R_2^6 = 2341.2$, and $R_3 = \Phi_3^3 + R_3^3 = 1736.2$.

Minimizing the maximum end-to-end response-time bound. For this choice, relative-deadline settings, obi-task response-time bounds, and obi-task offsets are as shown in Table 6.2 (b). The resulting end-to-end response-time bounds are $R_1 = \Phi_1^4 + R_1^4 = 2650.4$, $R_2 = \Phi_2^6 + R_2^6 = 2650.4$, and $R_3 = \Phi_3^3 + R_3^3 = 2650.4$.

Minimizing the maximum proportional end-to-end response-time bound. For this choice, relative-deadline settings, obi-task response-time bounds, and obi-task offsets are as shown in Table 6.2 (c). The resulting end-to-end response-time bounds are $R_1 = \Phi_1^4 + R_1^4 = 2208.9$, $R_2 = \Phi_2^6 + R_2^6 = 4417.8$, and $R_3 = \Phi_3^3 + R_3^3 = 4261.0$.

Early releasing. As discussed in Section 6.6, early releasing can improve observed response times without compromising response-time bounds. The value of allowing early releasing can be seen in the results reported

D_1^1	D_1^2	D_1^3	$\boldsymbol{D_1^4}$	D_2^1	D_2^2	D_2^3	D_2^4	D_2^5	$\boldsymbol{D_2^6}$	D_3^1	D_3^2	$\boldsymbol{D_3^3}$
500	500	500	**500**	0	0	0	0	772.84	**0**	0	0	**0**
R_1^1	R_1^2	R_1^3	$\boldsymbol{R_1^4}$	R_2^1	R_2^2	R_2^3	R_2^4	R_2^5	$\boldsymbol{R_2^6}$	R_3^1	R_3^2	$\boldsymbol{R_3^3}$
1034.4	1015.7	984.36	**1084.4**	579.36	558.5	592	611.36	1203.4	**0**	549.36	671.5	**515.36**
Φ_1^1	Φ_1^2	Φ_1^3	$\boldsymbol{\Phi_1^4}$	Φ_2^1	Φ_2^2	Φ_2^3	Φ_2^4	Φ_2^5	$\boldsymbol{\Phi_2^6}$	Φ_3^1	Φ_3^2	$\boldsymbol{\Phi_3^3}$
0	1034.4	1049.2	**2050.1**	0	579.36	1137.9	1729.9	1137.9	**2341.2**	0	549.36	**1220.9**

(a)

D_1^1	D_1^2	D_1^3	$\boldsymbol{D_1^4}$	D_2^1	D_2^2	D_2^3	D_2^4	D_2^5	$\boldsymbol{D_2^6}$	D_3^1	D_3^2	$\boldsymbol{D_3^3}$
0	500	359.06	**500**	0	0	0	584.52	1000	**0**	505.63	1000	**0**
R_1^1	R_1^2	R_1^3	$\boldsymbol{R_1^4}$	R_2^1	R_2^2	R_2^3	R_2^4	R_2^5	$\boldsymbol{R_2^6}$	R_3^1	R_3^2	$\boldsymbol{R_3^3}$
642.06	894.75	894.75	**1113.6**	608.56	437.5	471	1133.3	1424.1	**0**	1004.8	1101	**544.56**
Φ_1^1	Φ_1^2	Φ_1^3	$\boldsymbol{\Phi_1^4}$	Φ_2^1	Φ_2^2	Φ_2^3	Φ_2^4	Φ_2^5	$\boldsymbol{\Phi_2^6}$	Φ_3^1	Φ_3^2	$\boldsymbol{\Phi_3^3}$
0	642.06	642.06	**1536.8**	0	608.56	1046.1	1517.1	1128.8	**2650.4**	0	1004.8	**2105.8**

(b)

D_1^1	D_1^2	D_1^3	$\boldsymbol{D_1^4}$	D_2^1	D_2^2	D_2^3	D_2^4	D_2^5	$\boldsymbol{D_2^6}$	D_3^1	D_3^2	$\boldsymbol{D_3^3}$
0	0	200.53	**0**	1000	0	253.21	1000	1000	**0**	1000	1000	**1000**
R_1^1	R_1^2	R_1^3	$\boldsymbol{R_1^4}$	R_2^1	R_2^2	R_2^3	R_2^4	R_2^5	$\boldsymbol{R_2^6}$	R_3^1	R_3^2	$\boldsymbol{R_3^3}$
679.95	798.99	798.99	**729.95**	1489.4	616.99	789.89	1521.4	1461.9	**0**	1459.4	1280.5	**1425.4**
Φ_1^1	Φ_1^2	Φ_1^3	$\boldsymbol{\Phi_1^4}$	Φ_2^1	Φ_2^2	Φ_2^3	Φ_2^4	Φ_2^5	$\boldsymbol{\Phi_2^6}$	Φ_3^1	Φ_3^2	$\boldsymbol{\Phi_3^3}$
0	679.95	679.95	**1478.9**	0	1489.4	2106.4	2896.3	2552.7	**4417.8**	0	1508.3	**2835.6**

(c)

Table 6.2: Case-study relative-deadline settings, obi-task response-time bounds, and obi-task offsets when using linear programming to **(a)** minimize average end-to-end response-time bounds, **(b)** minimize maximum end-to-end response-time bounds, and **(c)** minimize maximum proportional end-to-end response-time bounds. Bold entries denote sinks.

in Table 6.3. This table gives the largest observed end-to-end response time of each DAG in Figure 6.7, assuming implicit deadlines with and without early releasing, in a schedule that was simulated for 50,000 time units. Analytical bounds are shown as well.

	G_1	G_2	G_3
Early releasing	1006	897	453
No early releasing	1966.75	3536.25	2586.0
Bounds	2538.25	4361.5	3376.5

Table 6.3: Observed end-to-end response times with/without early releasing and analytical end-to-end response-time bounds for the implicit-deadline setting.

6.8 Schedulability Studies

In this section, we expand upon the specific case study just described by considering general schedulability trends seen in experiments involving randomly generated task systems.

6.8.1 Improvements Enabled by Basic Techniques

We first consider the improvements enabled by the basic techniques covered in Sections 6.3 and 6.4 that underlie our work: allowing intra-task parallelism as provided by the npc-sporadic task model, and determining relative-deadline settings by solving an LP.

Random system generation. In our experiments, we considered a heterogeneous platform comprised of three CE pools, each consisting of eight identical CEs. Each pool was assumed to have the same total utilization. We considered all choices of total per-pool utilizations in the range $[1,8]$ in increments of 0.5.

We generated DAG-based task systems using a method similar to that used by others (Baruah, 2014; Li et al., 2013). These systems were generated by first specifying the number of DAGs in the system, N, and the number of tasks per DAG, n. For each considered pair N and n, we randomly generated 50 *task-system structures*, each comprised of N DAGs with n nodes. Each node in such a structure was randomly assigned to one of the CE pools, and for each DAG in the structure, one node was designated as its source, and one as its sink. Further, each pair of internal nodes (not a source or a sink) was connected by an edge with probability `edgeProb`, a settable parameter. Such an edge was directed from the lower-indexed node to the higher-indexed node, to preclude cycles. Finally, an edge was added from the source to each internal node with no incoming edges, and to the sink from each internal node with no outgoing edges.

For each considered per-pool untilization and each generated task-system structure, we randomly generated 50 actual task systems by generating task utilizations using the MATLAB function `randfixedsum()` (Stafford, 2006). According to the application domain that motivates this work,[3] we defined each DAG's period to be 1 *ms*. (A task's WCET is determined by its utilization and period.) For each considered value of N, n, and total per-pool utilization (one point in one of our graphs), we considered 50 (task system structures) $\times$ 50 (utilizations) $= 2,500$ task sets.

Comparison setup. We compared three strategies: **(i)** transforming to a conventional sporadic task system and using implicit relative deadlines, which is a strategy used in prior work on identical platforms (Liu and Anderson, 2010); **(ii)** transforming to an npc-sporadic task system and using implicit relative deadlines; and **(iii)** transforming to an npc-sporadic task system and using LP-based relative deadlines. When applying our LP techniques, we chose the objective function that minimizes the maximum end-to-end response-time bound. Although an identical platform was assumed in (Liu and Anderson, 2010), the techniques from that paper can be extended to heterogeneous platforms in a similar way to this chapter.

Results. In all cases that we considered, the two evaluated techniques improved end-to-end response-time bounds, often significantly. Similar trends were observed in all experiments we conducted, and we present here only the case where $N = 5$, $n = 20$, and `edgeProb` $= 0.5$ for demonstration. For each generated task set, we recorded the maximum end-to-end response-time bound among its five DAGs. For each given total per-pool utilization point, we report here the average of the maximum end-to-end response-time bounds among the $2,500$ task sets generated for that point. We call this metric the *average maximum end-to-end response-time bound (AMERB)*. Figure 6.9 plots AMERBs as a function of total per-CE-pool utilization. As seen, the application of both techniques reduced AMERBs by 39.42% to 81.65%.

6.8.2 Improvements Enabled by DAG Combining

As mentioned in Section 6.5, in the application domain that motivates our work, DAGs are usually defined using several well-defined computational templates, and as a result, many identical DAGs will exist. We proposed the technique of DAG combining in Section 6.5 to exploit this fact to further reduce response-time bounds. We now discuss schedulability experiments that we conducted to evaluate this technique.

[3] In applications usually considered in the real-time-systems community, much larger periods are the norm. The considered domain is quite different.

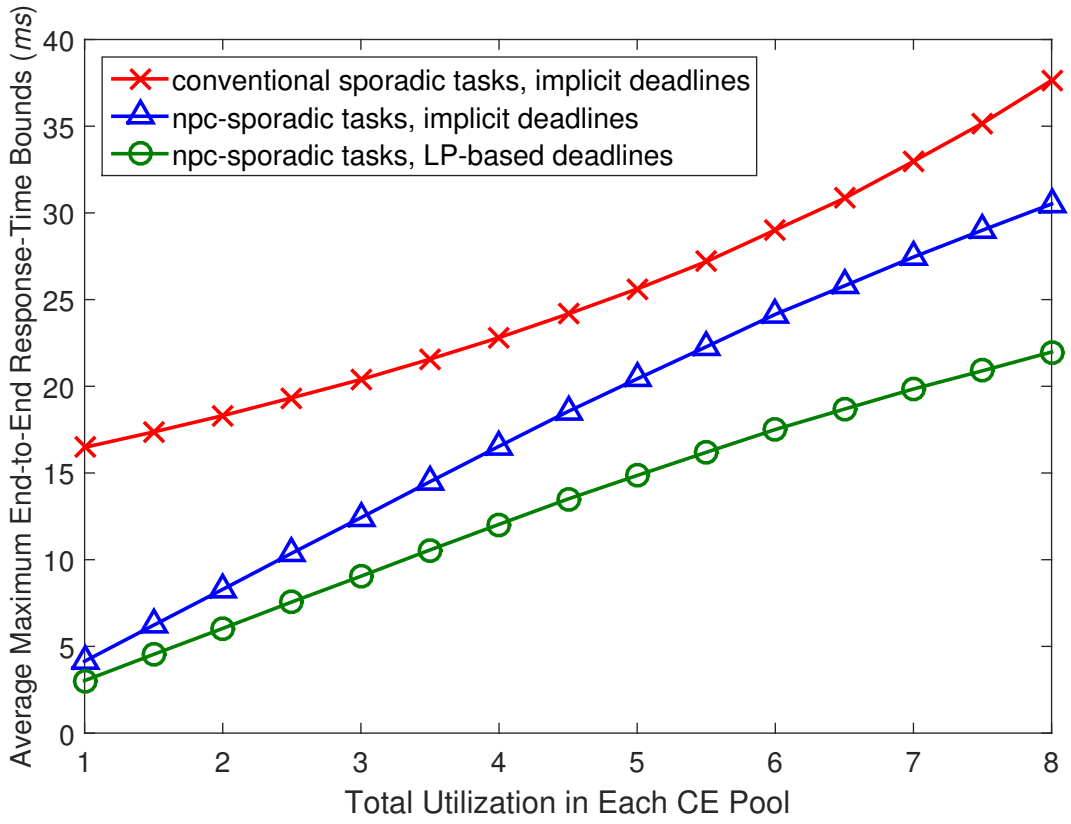


Figure 6.9: AMERBs as a function of total utilization in each CE pool in the case where each task set has five DAGs, 20 tasks per DAG, and `edgeProb`=0.5.

Random system generation. We employed a process of randomly generating systems that is similar to that discussed in Section 6.8.1, except that, instead of generating task-system structures comprised of N DAGs, we generated structures comprised of N *templates*. Additionally, we introduced a new parameter K that indicates the number of identical DAGs per template. A period of 1 *ms* was still associated with each DAG.

Comparison setup. We compared two strategies: **(i)** do no combining, and compute end-to-end response-time bounds assuming $N \cdot K$ independent DAGs; **(ii)** combine identical DAGs, and compute end-to-end response-time bounds assuming N DAGs, making adjustments as discussed in Section 6.5 to obtain actual response-time bounds for the DAGs that were combined. Under both strategies, the general techniques evaluated in Section 6.8.1 were applied.

Results. In all cases that we considered, the DAG combining technique improved end-to-end response-time bounds significantly. Similar trends were observed in all experiments we conducted, and we present here only the case where each system has five templates, each of which has 20 nodes, and $\texttt{edgeProb} = 0.5$ for demonstration. For this case, Figure 6.10 plots AMERBs as a function of total per-pool utilization, when the number of identical DAGs per template is fixed to 40 (this number is close to what would be expected in the application domain that motivates this work). Note that the AMERBs in Figure 6.10 are much lower than those in Figure 6.9, even before applying the DAG combining technique. This is because the systems considered in Figure 6.10 have far more DAGs than those in Figure 6.9. As a result, for each given total per-pool utilization, the systems in Figure 6.10 have much lower per-DAG and per-task utilizations. As also seen in Figure 6.10, when DAG combining is applied, the AMERBs are not very much influenced by the total per-CE-pool utilization. That is because DAG combining resulted in quite small response-time bounds by (6.9), so total end-to-end bounds were mainly impacted by the introduced shifting, rather than total per-CE-pool utilization. Also, Figure 6.11 plots AMERBs as a function of the number of identical DAGs per template, when every CE pool is fully utilized (*i.e.*, the total utilization of each pool is eight). In this case, the AMERB metric was calculated over all task sets that have the same number of identical DAGs per template.

According to our industry partners, in the considered application domain, a DAG's end-to-end response-time bound should typically be at most 2.35 *ms*. As observed in Figure 6.10, in the absence of DAG combining, AMERBs in this experiment were as high as 8.2 *ms*. However, the introduction of DAG combining enabled a drop to less than 2.0 *ms*, even when the platform was fully utilized. This demonstrates that DAG combining—as simple as it may seem—can have a powerful impact in the targeted domain.

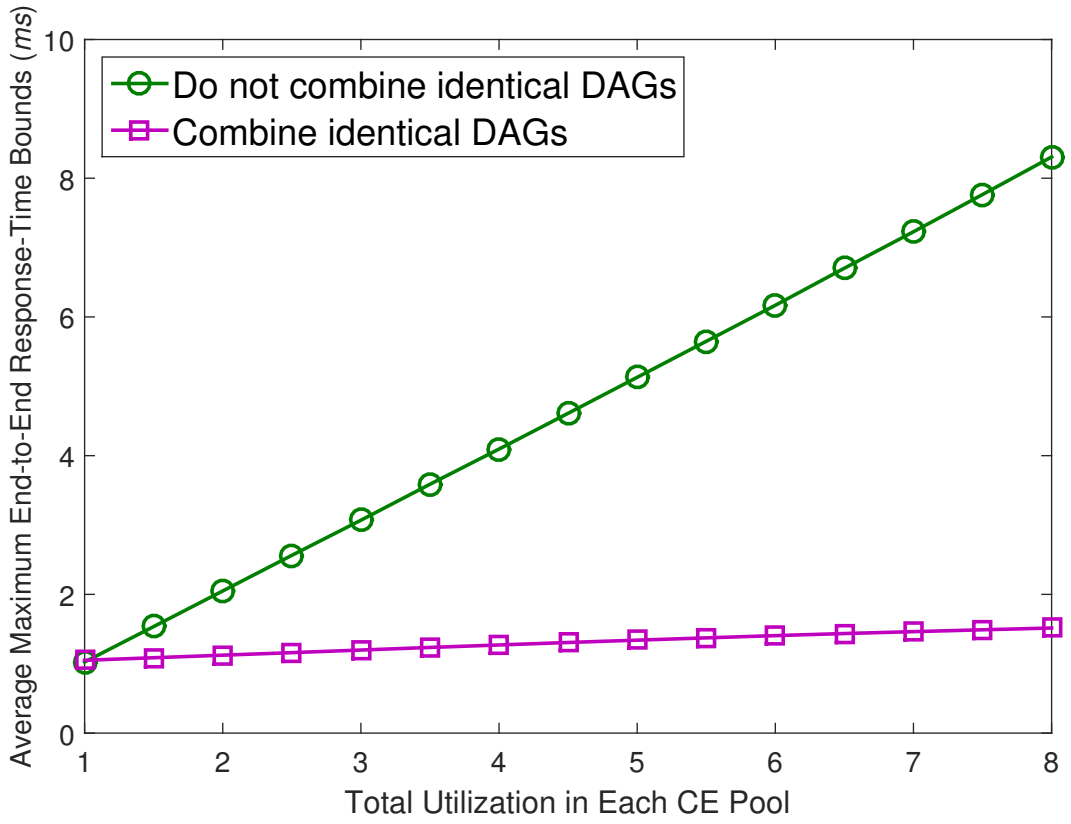


Figure 6.10: AMERBs as a function of total utilization in each CE pool in the case where the number of identical DAGs per template is fixed to 40.

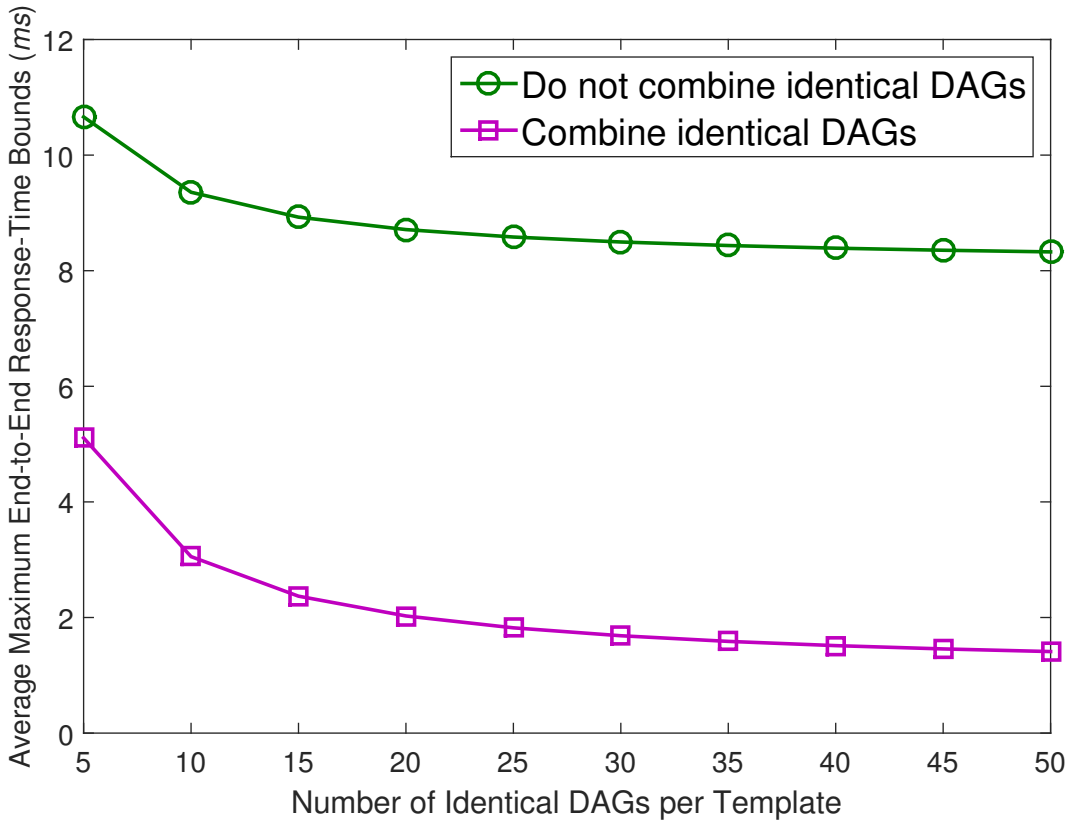


Figure 6.11: AMERBs as a function of the number of identical DAGs per template in the case where total utilization in each CE pool is fixed to eight.

6.9 Chapter Summary

In this chapter, we presented task-transformation techniques to provide end-to-end response-time bounds for DAG-based tasks implemented on heterogeneous multiprocessor platforms where intra-task parallelism is allowed. We also presented an LP-based method for setting relative deadlines and a DAG combining technique that can be applied to improve these bounds. We evaluated the efficacy of these results by considering a case-study task system and by conducting schedulability studies.

CHAPTER 7: MINIMUM-PARALLELISM MULTIPROCESSOR SUPPLY ON IDENTICAL PLATFORMS[1]

Open-systems (Deng and Liu, 1997) frameworks allow separate software components to execute together on a common hardware platform, with each component having the "illusion" of executing on a dedicated virtual platform. Providing such an illusion can ease software-development efforts, not only when mixing different applications, but also when integrating separately developed components of the same application. In domains where real-time constraints exist, *temporal isolation* among components should be ensured, *i.e.*, it should be possible to validate the timing constraints of each component independently. Therefore, a specification of the computing capacity allocated to a component is needed.

In early work in this direction pertaining to uniprocessor platforms, Shin and Lee (2003) proposed a virtual processor (VP) model called the *periodic resource (PR)* model, which allows the considerable body of work on periodic task scheduling (Liu and Layland, 1973) to be exploited in reasoning about the allocation of processor time to components. In the PR model, a VP is specified by the parameters (Π, Θ), with the interpretation that Θ time units of processor time is guaranteed to the supported component every Π time units.

While this simple model sufficed in the uniprocessor case, it is inadequate in the multiprocessor case, because the important issue of *parallelism* is ignored. To deal with this issue, Shin et al. (2008) proposed extending the PR model by adding an additional parameter. Specifically, under their *multiprocessor periodic resource (MPR)* model, the supply allocated to a component is specified by (Π, Θ, m'), with the interpretation that Θ time units of processor time is guaranteed to the component every Π time units with at most m' VPs providing allocation in parallel. That is, the new parameter m' specifies the *maximum degree of parallelism*. In the MPR model, all VPs allocated to a component are required to have a common period Π that is strictly synchronized.

[1]Contents of this chapter previously appeared in preliminary form in the following paper:

Yang, K. and Anderson, J. (2016a). On the dominance of minimum-parallelism multiprocessor supply. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 215–226.

A key characteristic of the MPR model is its flexibility. For example, consider a component that is to be allocated 80% of the capacity of a quad-core machine. The supply interface for that component could be defined as $(100, 320, 4)$, meaning that every 100 time units, the component receives 320 units of processing time on up to four processors. Such a specification does not indicate the precise manner in which processing time is allocated. For example, the component could be allocated 80% of the capacity of each processor, or 100% of three processors and 20% of the fourth, among other choices. Which choice is best?

MP form. In the example just discussed, the second-listed choice is known as *minimum-parallelism (MP) form*. Under MP form, each component is allocated at most one partially available processor, with all other processors allocated to it being fully available. MP form was first proposed by Leontyev and Anderson (2009) to support SRT container hierarchies, which allow components to include sub-components, which in turn can include their own sub-components, *etc.* Assuming MP form, they showed that container hierarchies with an unlimited number of levels can be supported with bounded deadline tardiness and no utilization loss. In work directed at HRT systems, Xu et al. (2015) observed that, by enforcing MP form in the context of the MPR model, per-component schedulability can be improved.

Because this improvement in schedulability was considered in the context of the MPR model, a common, synchronized allocation period was assumed to be used on all processors allocated to a component. In practice, however, situations exist in which such an assumption may be problematic. A good example of this can be seen in recent work of Durrieu et al. (2014), who considered a flight management system implemented on a multicore platform wherein clocks on different processors "do not drift [but] have unpredictable initial offsets." In the future, the assumption of tight synchrony may become even more problematic, as manycore platforms evolve in which core counts soar into the hundreds if not thousands. Similar observations have been made by Lipari and Bini (2010) and Bini et al. (2009b), who suggested generalizing the MPR model so that the VPs allocated to a single component may have different periods with different initial phasings. Does MP form still retain its advantages over other supply forms in the HRT case under this more general notion of VP allocation?

In chapter, we answer this question in the affirmative by showing that MP form dominates all other supply forms in the context of these cases: VPs are synchronous, concrete asynchronous, or non-concrete asynchronous (these terms are defined in Section 7.1). In each of these cases, we consider two sub-cases: requiring a common period for all VPs, and allowing such periods to differ. The prior work noted above by

	Common Period	Different Periods
Synchronous	Theorem 7.5	Theorem 7.6
Concrete Asynchronous	Theorem 7.6	Theorem 7.6
Non-Concrete Asynchronous	Theorem 7.2	Theorems 7.3 and 7.4

Table 7.1: Summary of theorems applying to different VP synchronization assumptions.

Xu et al. (2015) on the MPR model implies that MP form dominates all other forms in the case of synchronous VPs with a common period. For each other case, we show that an arbitrary component is always dominated by an MP-form component of the same bandwidth (*i.e.*, total processor capacity—see Section 7.1), provided its period is defined properly. These results follow from the theorems listed in Table 7.1. Additionally, in all six cases, we show that an MP-form component can never be dominated by a non-MP-form component of the same bandwidth, regardless of how periods are defined. The issue of MP dominance under the considered cases is not as straightforward as one might think at first glance. Indeed, many subtleties arise.

Organization. In the following sections, we introduce our system model (Section 7.1), provide some preliminary properties and theorems (Section 7.2), show the dominance of MP form for non-concrete asynchronous VPs (Section 7.3) and synchronous and concrete asynchronous VPs (Section 7.4), and show that MP form cannot be dominated by any other form (Section 7.5).

7.1 System Model

We consider a compositional system executing upon a physical multiprocessor platform with identical processors. Each component is provided processor time by a set of VPs, each defined according to the PR model, as discussed next.

7.1.1 Periodic Resource Model

Under the PR model (Shin and Lee, 2003), a VP Γ_i is characterized by two parameters (Π_i, Θ_i), which indicate that Γ_i supplies Θ_i units of processor time every Π_i time units, where $0 < \Theta_i \leq \Pi_i$. In this chapter, we assume continuous time, thus Π_i and Θ_i are real numbers. The *bandwidth* of the VP Γ_i is given by $w_i = \Theta_i/\Pi_i$. Note that, for any Π_i, $\Gamma_i = (\Pi_i, \Pi_i)$ defines a VP corresponding to a dedicated physical processor that is always available.

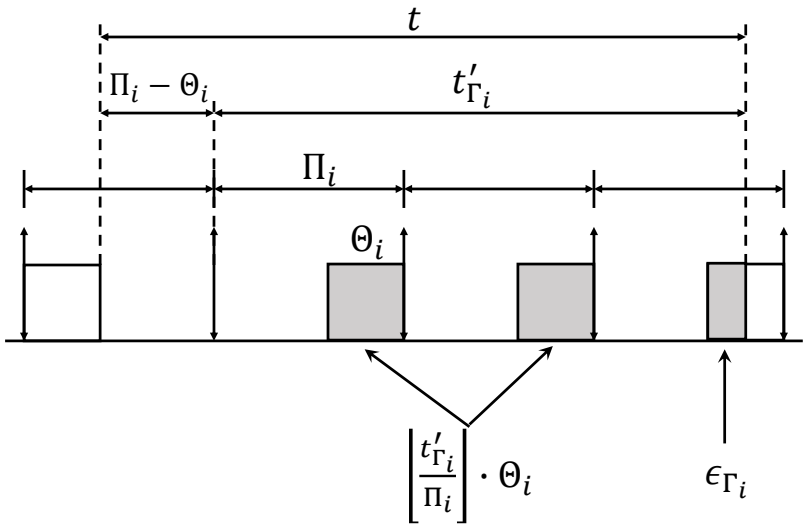


Figure 7.1: Worst-case supply of Γ_i (adapted from (Shin and Lee, 2003)).

The *supply bound function (SBF)* of the VP Γ_i, denoted $Z(t,\Gamma_i)$, indicates the *minimum* processor time Γ_i can supply during any time interval of length t. Shin and Lee (Shin and Lee, 2003) have shown that $Z(t,\Gamma_i)$ can be defined as

$$Z(t,\Gamma_i) = \begin{cases} 0 & \text{if } t'_{\Gamma_i} < 0 \\ \left\lfloor \frac{t'_{\Gamma_i}}{\Pi_i} \right\rfloor \cdot \Theta_i + \varepsilon_{\Gamma_i} & \text{if } t'_{\Gamma_i} \geq 0 \end{cases} \tag{7.1}$$

where

$$t'_{\Gamma_i} = t - (\Pi_i - \Theta_i), \tag{7.2}$$

$$\varepsilon_{\Gamma_i} = \max\left(t'_{\Gamma_i} - \Pi_i \left\lfloor \frac{t'_{\Gamma_i}}{\Pi_i} \right\rfloor - (\Pi_i - \Theta_i), 0\right). \tag{7.3}$$

This definition reflects the worst-case scenario illustrated in Figure 7.1.

7.1.2 VPs in a Component

We consider a component $\mathcal{C}$ that consists of a set of VPs, denoted $\mathcal{C} = \{\Gamma_i\}$, where $\Gamma_i = (\Pi_i, \Theta_i)$ for $1 \leq i \leq |\mathcal{C}|$. The supply of a component is the sum of the supply of all VPs in this component.

Since $\Gamma_i = (\Pi_i, \Pi_i)$ indicates a dedicated processor regardless of the value of Π_i, we let p denote the number of such dedicated processors and do not bother to specify their periods. Thus, we alternatively denote

the component $\mathcal{C}$ by $\mathcal{C} = (p, \mathcal{T})$, where $\mathcal{T} = \{\Gamma_i \mid \Gamma_i \in \mathcal{C} \wedge 0 < w_i < 1\}$. It is clear that

$$|\mathcal{C}| = p + |\mathcal{T}|. \tag{7.4}$$

We define the *bandwidth* of component $\mathcal{C}$ as

$$\mathsf{bw}(\mathcal{C}) = \sum_{\Gamma_i \in \mathcal{C}} w_i. \tag{7.5}$$

The bandwidth $\mathsf{bw}(\mathcal{C})$ indicates the total processor share allocation to which $\mathcal{C}$ is entitled. *Minimum-parallelism (MP)* form is defined as follows.

Definition 7.1. A component $\mathcal{C} = (p, \mathcal{T})$ is in MP form if and only if $|\mathcal{T}| \leq 1$.

Concrete vs. non-Concrete. We consider the possibility that the VPs in a component are *asynchronous*, meaning that they can have different phases—a VP Γ_i with a *phase* of ϕ_i is initialized to begin at time ϕ_i, *i.e.*, its first allocation of Θ_i time units occurs within the interval $[\phi_i, \phi_i + \Pi_i)$, its second within $[\phi_i + \Pi_i, \phi_i + 2\Pi_i)$, and so on. As it turns out, the results we obtain depend on whether phases are *known* or *unknown* prior to runtime. In the first case, we say that the VPs are *concrete* asynchronous, and only a particular phase for each VP needs to be considered in schedulability (supply) analysis. In the second case, we say that the VPs are *non-concrete* asynchronous, and the worst case among all possible phases must be considered in schedulability (supply) analysis . *Synchronous* VPs can be considered as a special case of concrete asynchronous VPs where all phases are required to be zero. In this chapter, we consider all of the three phasing assumptions regarding VPs: they can be synchronous, concrete asynchronous, or non-concrete asynchronous.

7.1.3 Parallel Supply Function

The SBF definition in (7.1) for the PR model hinges only on considering uniprocessor supply allocations. In the multiprocessor case, however, SBFs must also address the important issue of parallelism. Various multiprocessor SBFs have been proposed. The most expressive of these considered to date is the *parallel supply function (PSF)*, proposed by Bini et al. (2009a). The PSF describes the supply of a component $\mathcal{C}$ by a set of functions, $\{\mathsf{psf}_j(t, \mathcal{C}) \mid j \in \mathbb{Z}^+\}$, where each function $\mathsf{psf}_j(t, \mathcal{C})$ is defined as follows.

Definition 7.2. $\mathsf{psf}_j(t, \mathcal{C})$ denotes the *minimum* supply of $\mathcal{C}$ during *any time interval* of length t with a degree of parallelism at most j.

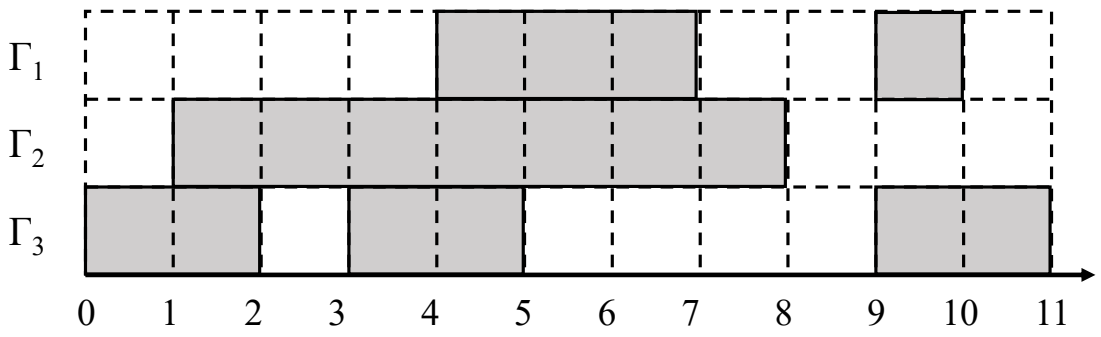


Figure 7.2: Example illustrating parallel supply (adapted from (Lipari and Bini, 2010)).

We illustrate the above definition with the following example, and refer readers to the work of Bini et al. (2009a) for a more formal treatment.

Example 7.1. (Adapted from Lipari and Bini (2010).) Let Γ_1, Γ_2,, and Γ_3 be three VPs that compose $\mathcal{C}$. Assume that the processor time they make available within the time interval $[0,11)$ is shown in Figure 7.2, where the gray boxes represent available processor time. Suppose that all three VPs are fully available at or after time 11. Then, $[0,11)$ is the interval of length 11 that provides the minimum supply at every degree of parallelism. In this case, $\mathsf{psf}_1(t,\mathcal{C}) = 10$ because there are 10 time units in $[0,11)$ during which at least one VP provides available processor time. $\mathsf{psf}_2(t,\mathcal{C}) = 16$ because all three VPs provide available processor time simulanteously only in $[4,5)$, so $\mathsf{psf}_2(t,\mathcal{C})$ is one less than the total available processor time in $[0,11)$. This total available time is given by $\mathsf{psf}_3(t,\mathcal{C}) = 17$. $\Diamond$

In this chapter, we use PSF functions to describe *exact* lower bounds on supply in order to compare the supply of different components *exactly*. That is, for any j and $t \geq 0$, there exists a possible scenario in which, over some interval of length t, the supply provided by $\mathcal{C}$ with a degree of parallelism at most j is exactly $\mathsf{psf}_j(t,\mathcal{C})$.

By Def. 7.2, we have the following property.

$$(\forall \mathcal{C}, \forall j \geq 1, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq jt) \tag{7.6}$$

Also, By Lemma 1 in (Bini et al., 2009a), the following properties hold.

$$(\forall \mathcal{C}, \forall j \geq 1, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq \mathsf{psf}_{j+1}(t,\mathcal{C})) \tag{7.7}$$

$$(\forall \mathcal{C}, \forall j \geq |\mathcal{C}|, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) = \mathsf{psf}_{j+1}(t,\mathcal{C})) \tag{7.8}$$

In accordance with Def. 7.2, $\mathsf{psf}_\infty(t,\mathcal{C})$ represents the minimum supply that $\mathcal{C}$ is guaranteed to provide during any time interval of length t *with no constraint on the degree of parallelism*. By Def. 7.2, $\mathsf{psf}_\infty(t,\mathcal{C}) = \mathsf{psf}_{|\mathcal{C}|}(t,\mathcal{C})$, because there are at most $|\mathcal{C}|$ dedicated or non-dedicated resources that can provide supply in parallel in $\mathcal{C}$.

7.2 Preliminaries

In this section, we provide a condition for establishing the superiority of MP form. This condition will allow us to conclude that MP form dominates other forms. Dominance is defined with respect to component supply based on PSF:

Definition 7.3. A component $\mathcal{C}'$ *dominates* another component $\mathcal{C}$ if and only if $(\forall j \geq 1, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq \mathsf{psf}_j(t,\mathcal{C}'))$ holds.

By Def. 7.3, in order to show the dominance of an arbitrary component $\mathcal{C}'$ over another arbitrary component $\mathcal{C}$, we must consider all relevant PSF functions. However, the following theorem shows that it suffices to consider only two specific PSF functions.

Theorem 7.1. *Let $\mathcal{C}$ be an arbitrary component, and let $\mathcal{C}^*$ be a component in MP form. If $(\forall t :: \mathsf{psf}_\infty(t,\mathcal{C}) \leq \mathsf{psf}_\infty(t,\mathcal{C}^*))$ holds, then $\mathcal{C}^*$ dominates $\mathcal{C}$.*

Proof. Let $\mathcal{C} = (p, \mathcal{T})$ and $\mathcal{C}^* = (p^*, \mathcal{T}^*)$. Because $\mathcal{C}^*$ has p^* dedicated processors,

$$(\forall 1 \leq j \leq p^*, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}^*) = jt). \tag{7.9}$$

On the other hand, for $\mathcal{C}$, by (7.6), we have

$$(\forall 1 \leq j \leq p^*, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq jt). \tag{7.10}$$

By (7.9) and (7.10),

$$(\forall 1 \leq j \leq p^*, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq \mathsf{psf}_j(t,\mathcal{C}^*)). \tag{7.11}$$

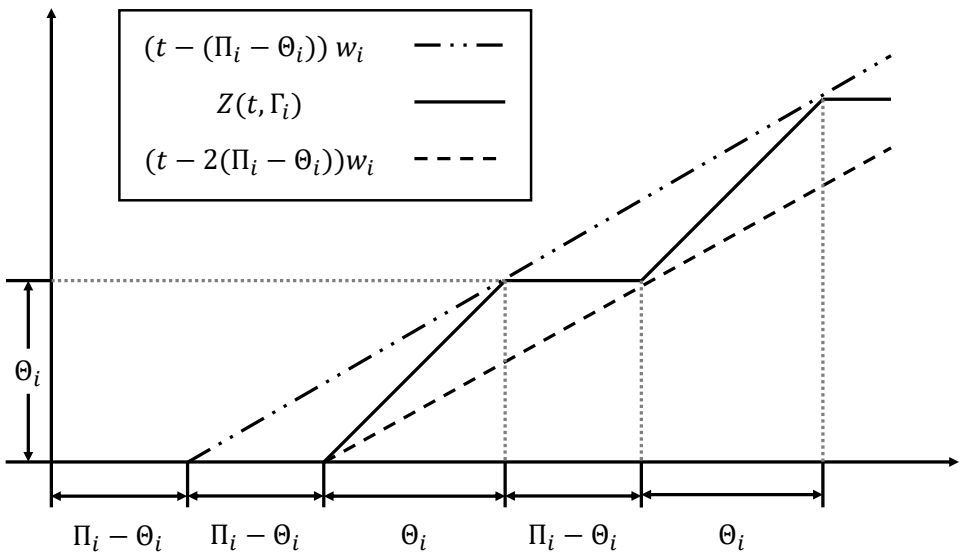


Figure 7.3: The graph of $Z(t,\Gamma_i)$, as an illustration of Properties 7.1, 7.2, and 7.3.

Because $\mathcal{C}^*$ is in MP form, $|\mathcal{T}| \leq 1$, and by (7.4), $|\mathcal{C}^*| = p^* + |\mathcal{T}^*| \leq p^* + 1$. Therefore, by (7.8),

$$(\forall j \geq p^* + 1, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}^*) = \mathsf{psf}_\infty(t,\mathcal{C}^*)). \tag{7.12}$$

On the other hand, for $\mathcal{C}$, by (7.7),

$$(\forall j \geq p^* + 1, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq \mathsf{psf}_\infty(t,\mathcal{C})). \tag{7.13}$$

Now, by (7.12), (7.13), and $\mathsf{psf}_\infty(t,\mathcal{C}) \leq \mathsf{psf}_\infty(t,\mathcal{C}^*)$ (from the statement of the theorem), we have

$$(\forall j \geq p^* + 1, \forall t \geq 0 :: \mathsf{psf}_j(t,\mathcal{C}) \leq \mathsf{psf}_j(t,\mathcal{C}^*)). \tag{7.14}$$

By (7.11), (7.14), and Def. 7.3, $\mathcal{C}^*$ dominates $\mathcal{C}$. □

Before endeavoring to use Theorem 7.1 to establish the dominance of MP form, we first provide several useful properties concerning the supply function $Z(t,\Gamma_i)$ of an arbitrary VP Γ_i. Property 7.1 directly follows from the definition of $Z(t,\Gamma_i)$ as given by (7.1)–(7.3). Property 7.2 is established in Lemma 1 in (Shin and Lee, 2003), and Property 7.3 is established in (Easwaran et al., 2007). The intuition behind these properties is illustrated by the graph of $Z(t,\Gamma_i)$ shown in Figure 7.3.

Property 7.1. $Z(t,\Gamma_i) = 0$ *for* $0 \le t \le 2(\Pi_i - \Theta_i)$.

Property 7.2. $Z(t,\Gamma_i) \ge \max\{(t - 2(\Pi_i - \Theta_i))w_i, 0\}$.

Property 7.3. $Z(t,\Gamma_i) \le \max\{(t - (\Pi_i - \Theta_i))w_i, 0\}$.

We state two more properties below, in which an alternate definition of $Z(t,\Gamma_i)$ is indirectly considered that is based on the following function f:

$$f(x,\Gamma_i) = \left\lfloor \frac{x}{\Pi_i} \right\rfloor \cdot \Theta_i + \max\left(x - \Pi_i \left\lfloor \frac{x}{\Pi_i} \right\rfloor - (\Pi_i - \Theta_i), 0 \right). \tag{7.15}$$

Note that, by (7.1) (7.2) and (7.3),

$$Z(t,\Gamma_i) = f(t'_{\Gamma_i},\Gamma_i), \text{ if } t'_{\Gamma_i} \ge 0. \tag{7.16}$$

When Γ_i is fixed, *i.e.*, Π_i and Θ_i are constants, the following properties apply to $f(x,\Gamma_i)$. These properties can be seen intuitively by considering the graph of $f(x,\Gamma_i)$, which is similar to that of $Z(t,\Gamma_i)$ as illustrated in Figure 7.3. Property 7.5 can be seen by observing that the slope of any two points in the graph of $f(x,\Gamma_i)$ is at most one.

Property 7.4. *$f(x,\Gamma_i)$ is monotonically increasing for non-negative x, i.e., $f(x_1,\Gamma_i) \le f(x_2,\Gamma_i)$ if $0 \le x_1 \le x_2$.*

Property 7.5. *For any $x, y \ge 0$, $f(x+y,\Gamma_i) \le f(x,\Gamma_i) + y$, which also implies $f(x-y,\Gamma_i) \ge f(x,\Gamma_i) - y$, provided that $x - y \ge 0$ holds.*

We also utilize the two straightforward claims below.

Claim 7.1. *The supply of a VP Γ_i can be zero within* any *time interval of length $\Pi_i - \Theta_i$, regardless of how the interval aligns with the VP's periods of allocation.*

This claim is different from Property 7.1. In order to have a supply of zero within a time interval of length up to $2(\Pi_i - \Theta_i)$, as stated in Property 7.1, the interval must have a specific alignment with respect to the periods of allocation of Γ_i as shown in Figure 7.1. However, according to this claim, the supply within *any* time interval of length $\Pi_i - \Theta_i$ can be a zero. Figure 7.4 shows the only two possibilities that can occur: the considered interval is either included within a single period of allocation, or spans two such periods. In either situation, supply within the interval can be zero.

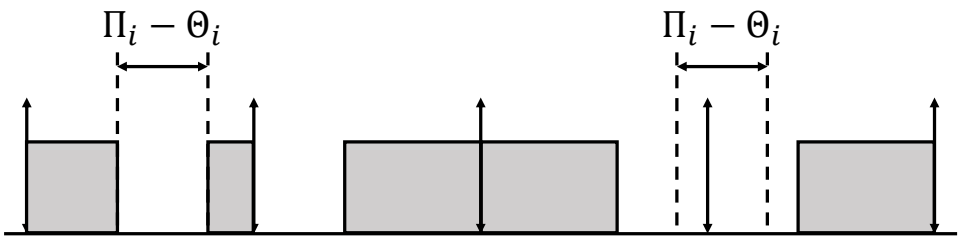


Figure 7.4: Illustration of Claim 7.1.

Claim 7.2. *Let $\mathcal{C}^* = (p^*, \mathcal{T}^*)$ be a component in MP form. If $|\mathcal{T}^*| = 0$, then $\mathsf{psf}_\infty(t, \mathcal{C}^*) = t \cdot p^*$. If $|\mathcal{T}^*| = 1$, then letting Γ^* denote the lone VP in $\mathcal{T}^*$, $\mathsf{psf}_\infty(t, \mathcal{C}^*) = t \cdot p^* + Z(t, \Gamma^*)$*

This claim follows directly from the definitions above.

7.3 Non-Concrete Asynchronous

In this section, we consider the case of *non-concrete asynchronous* VPs. In order to apply Theorem 7.1 in this case to establish the dominance of MP form, we begin by providing an *exact* calculation of $\mathsf{psf}_\infty(t, \mathcal{C})$.

For any time interval of length t, a dedicated resource supplies t time units of processor time, and by (7.1), a non-dedicated resource Γ supplies at least $Z(t, \Gamma)$ time units. Therefore, with the degree of parallelism unconstrained, a component $\mathcal{C} = (p, \mathcal{T})$ provides a supply of at least $tp + \sum_{\Gamma_i \in \mathcal{T}} Z(t, \Gamma_i)$. Moreover, this minimum does indeed happen, as shown in Figure 7.5. (Note that the alignment shown in the figure can happen because we are assuming for now that VPs are non-concrete asynchronous.) Thus, for any component $\mathcal{C} = (p, \mathcal{T})$,

$$\mathsf{psf}_\infty(t, \mathcal{C}) = tp + \sum_{\Gamma_i \in \mathcal{T}} Z(t, \Gamma_i). \tag{7.17}$$

In the next two subsections, we establish the dominance of MP form in two steps. First, we consider the case in which all VPs in $\mathcal{C}$ share a common period. Second, we build upon this result by considering the case in which the VPs in $\mathcal{C}$ may have different periods.

7.3.1 A Common Period

We first consider the case in which the VPs in $\mathcal{C}$ share a common period Π, *i.e.*, $(\forall \Gamma_i = (\Pi_i, \Theta_i) \in \mathcal{C} :: \Pi_i = \Pi)$ holds. We establish our key proof obligation in Theorem 7.2 below. The following lemma is used in its proof. Specifically, we use it to show how to combine two VPs "locally" in a way that is in accordance

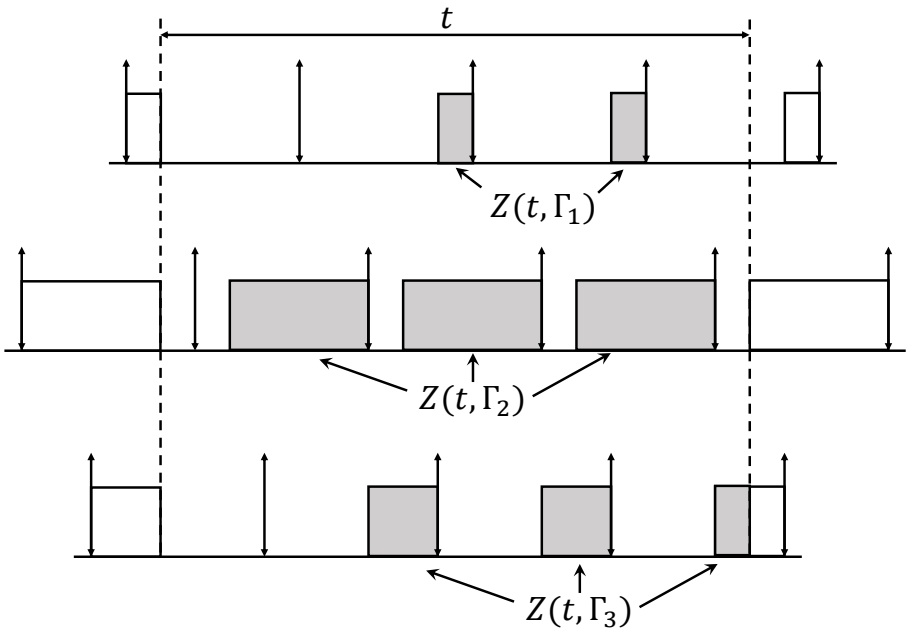


Figure 7.5: Illustration of the worst case of $\mathsf{psf}_\infty(t,\mathcal{C})$ for non-concrete asynchronous VPs.

with MP form. Figure 7.6 illustrates the three cases of the lemma. A rigorous proof is rather tedious and mechanical. Readers who are not interested may skip these mathematical details to Page 159.

Lemma 7.1. *Let $\Gamma_i = (\Pi, \Theta_i)$ and $\Gamma_j = (\Pi, \Theta_j)$ be two VPs that are not dedicated processors, and without loss of generality, assume $\Theta_i \le \Theta_j$, i.e., $0 < w_i \le w_j < 1$. Then, we have the following three exhaustive cases for $w_i + w_j$ and corresponding conclusions.*

1. *If $0 < w_i + w_j < 1$, then $Z(t,\Gamma_i) + Z(t,\Gamma_j) \le Z(t,\Gamma_k)$, where $\Gamma_k = (\Pi, \Theta_k)$ and $\Theta_k = \Theta_i + \Theta_j$.*
2. *If $w_i + w_j = 1$, then $Z(t,\Gamma_i) + Z(t,\Gamma_j) \le t$.*
3. *If $1 < w_i + w_j < 2$, then $Z(t,\Gamma_i) + Z(t,\Gamma_j) \le t + Z(t,\Gamma_k)$, where $\Gamma_k = (\Pi, \Theta_k)$ and $\Theta_k = \Theta_i + \Theta_j - \Pi$.*

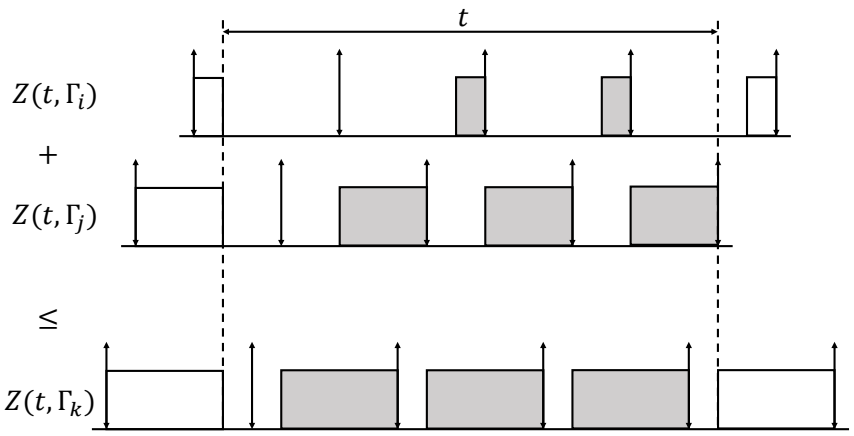
Proof. We consider the three cases of the lemma individually.

Case 1: In this case,

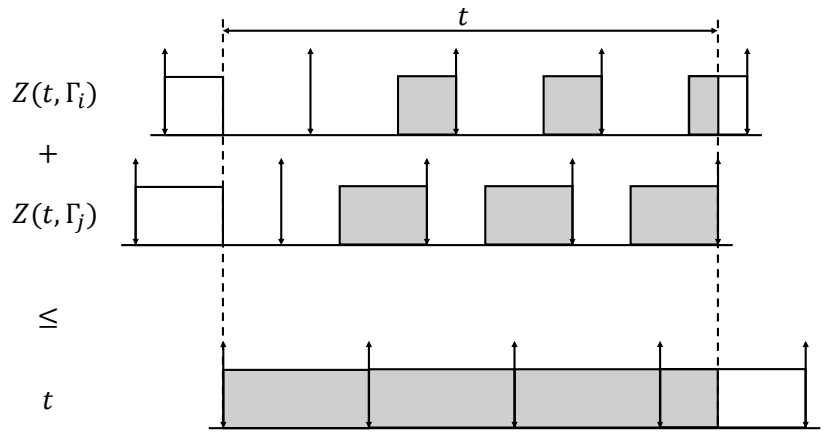
$$\Theta_k = \Theta_i + \Theta_j. \tag{7.18}$$

so

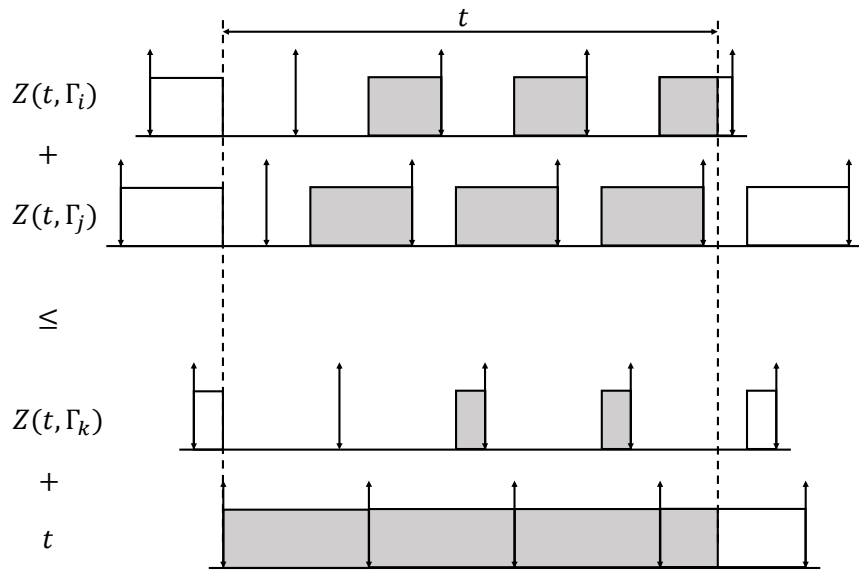
$$\Theta_i \le \Theta_j < \Theta_k, \tag{7.19}$$



(a) Illustration for Case 1



(b) Illustration for Case 2



(c) Illustration for Case 3

Figure 7.6: Illustration for the cases in Lemma 7.1.

because $\Theta_i \leq \Theta_j$ is assumed by the statement of the lemma. By (7.2), (7.18), and (7.19),

$$t'_{\Gamma_i} \leq t'_{\Gamma_j} < t'_{\Gamma_k}. \tag{7.20}$$

In the next paragraph, we dispense with all possibilities that occur when at least one of t'_{Γ_i} and t'_{Γ_j} is negative.

First, if $t'_{\Gamma_i} \leq t'_{\Gamma_j} < 0$, then by (7.1), $Z(t,\Gamma_i)+Z(t,\Gamma_j) = 0 \leq Z(t,\Gamma_k)$. Second, if $t'_{\Gamma_i} < 0 \leq t'_{\Gamma_j}$, then by (7.1), $Z(t,\Gamma_i)+Z(t,\Gamma_j) = 0+Z(t,\Gamma_j) \leq Z(t,\Gamma_k)$. Therefore, in the rest of the proof for Case 1, we focus on the remaining possibility, $0 \leq t'_{\Gamma_i} \leq t'_{\Gamma_j}$, which by (7.20), implies

$$0 \leq t'_{\Gamma_i} \leq t'_{\Gamma_j} < t'_{\Gamma_k}. \tag{7.21}$$

Applying (7.16) to $Z(t,\Gamma_i)$, $Z(t,\Gamma_j)$, and $Z(t,\Gamma_k)$, respectively, we have the following.

$$\begin{aligned}
Z(t,\Gamma_i) &= \{\text{by (7.16)}\} \\
&\quad f(t'_{\Gamma_i},\Gamma_i) \\
&\leq \{\text{by (7.21) and Property 7.4 }\} \\
&\quad f(t'_{\Gamma_k},\Gamma_i) \\
&= \{\text{by (7.15)}\} \\
&\quad \left\lfloor \frac{t'_{\Gamma_k}}{\Pi} \right\rfloor \Theta_i + \max\left(t'_{\Gamma_k} - \Pi \left\lfloor \frac{t'_{\Gamma_k}}{\Pi} \right\rfloor - (\Pi - \Theta_i), 0\right).
\end{aligned} \tag{7.22}$$

Similarly, for the same reasons,

$$Z(t,\Gamma_j) \leq \left\lfloor \frac{t'_{\Gamma_k}}{\Pi} \right\rfloor \Theta_j + \max\left(t'_{\Gamma_k} - \Pi \left\lfloor \frac{t'_{\Gamma_k}}{\Pi} \right\rfloor - (\Pi - \Theta_j), 0\right). \tag{7.23}$$

By (7.15) and (7.16),

$$Z(t,\Gamma_k) = \left\lfloor \frac{t'_{\Gamma_k}}{\Pi} \right\rfloor \Theta_k + \max\left(t'_{\Gamma_k} - \Pi \left\lfloor \frac{t'_{\Gamma_k}}{\Pi} \right\rfloor - (\Pi - \Theta_k), 0\right). \tag{7.24}$$

For notational simplicity, we introduce the two terms below.

$$\Phi = t'_{\Gamma_k} - \Pi \left\lfloor t'_{\Gamma_k}/\Pi \right\rfloor \tag{7.25}$$

$$\Delta = \max(\Phi - (\Pi - \Theta_i), 0) + \max(\Phi - (\Pi - \Theta_j), 0) - \max(\Phi - (\Pi - \Theta_k), 0) \tag{7.26}$$

Now, by (7.30), (7.31) and (7.32), we have

$$\begin{aligned} Z(t,\Gamma_i) + Z(t,\Gamma_j) - Z(t,\Gamma_k) &\leq \left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor (\Theta_i + \Theta_j - \Theta_k) + \Delta \\ &= \{\text{by (7.18)}\} \\ &\Delta. \end{aligned} \tag{7.27}$$

Given the derivation above, we can complete the proof by showing $\Delta \leq 0$. This result is implied by the following claim.

Claim 7.3. *In Case 1,* $\Delta \leq 0$.

Proof. By (7.25), $0 \leq \Phi < \Pi$. Also, by (7.19), $\Pi - \Theta_k < \Pi - \Theta_j \leq \Pi - \Theta_i$. Given these ranges, the following cases are exhaustive.

Case 1.1: $\Phi \in [0, \Pi - \Theta_k)$, which implies $\Delta = 0$.

Case 1.2: $\Phi \in [\Pi - \Theta_k, \Pi - \Theta_j)$, which implies $\Delta = 0 - (\Phi - (\Pi - \Theta_k)) \leq 0$, because $\Phi \geq \Pi - \Theta_k$ holds in this case.

Case 1.3: $\Phi \in [\Pi - \Theta_j, \Pi - \Theta_i)$, which implies $\Delta = (\Phi - (\Pi - \Theta_j)) - (\Phi - (\Pi - \Theta_k)) = \Theta_j - \Theta_k < 0$, by (7.19).

Case 1.4: $\Phi \in [\Pi - \Theta_i, \Pi)$, which implies $\Delta = (\Phi - (\Pi - \Theta_i)) + (\Phi - (\Pi - \Theta_j)) - (\Phi - (\Pi - \Theta_k)) = \Phi - \Pi + \Theta_i + \Theta_j - \Theta_k < 0$, by (7.18) and the fact that $\Phi < \Pi$ holds in this case. □

Claim 7.3 and (7.27) together imply $Z(t,\Gamma_i) + Z(t,\Gamma_j) \leq t + Z(t,\Gamma_k)$, as required.

Case 2: In this case, $w_i + w_j = 1$. By Property 7.3, $Z(t,\Gamma_i) \leq \max\{(t - (\Pi - \Theta_i))w_i, 0\} \leq tw_i$. Similarly, $Z(t,\Gamma_j) \leq tw_j$. Thus, $Z(t,\Gamma_i) + Z(t,\Gamma_j) \leq t(bw_i + bw_j) = t$.

Case 3: In this case,

$$\Theta_k = \Theta_i + \Theta_j - \Pi, \tag{7.28}$$

so

$$\Theta_k < \Theta_i \leq \Theta_j, \tag{7.29}$$

since $\Theta_j < \Pi$ holds and $\Theta_i \leq \Theta_j$ is assumed by the statement of the lemma. By (7.2), (7.28), and (7.29), $t'_{\Gamma_k} < t'_{\Gamma_i} \leq t'_{\Gamma_j}$. In the next paragraph, we dispense with all possibilities that occur when at least one of t'_{Γ_k}, t'_{Γ_i}, and t'_{Γ_j} is negative.

First, if $t'_{\Gamma_k} < t'_{\Gamma_i} \leq t'_{\Gamma_j} < 0$, then by (7.1), $Z(t,\Gamma_i) + Z(t,\Gamma_j) = 0 \leq t + 0 = t + Z(t,\Gamma_k)$. Second, if $t'_{\Gamma_k} < t'_{\Gamma_i} < 0 \leq t'_{\Gamma_j}$, then by (7.1), $Z(t,\Gamma_i) + Z(t,\Gamma_j) = 0 + Z(t,\Gamma_j) \leq 0 + t = t + 0 = t + Z(t,\Gamma_k)$. Third, if $t'_{\Gamma_k} < 0 \leq t'_{\Gamma_i} \leq t'_{\Gamma_j}$, then we have $t'_{\Gamma_k} < 0$, which by (7.2), implies $t - (\Pi - \Theta_k) < 0$, and hence, $t < \Pi - \Theta_k$. By the statement of the lemma (and in particular, Case 3), $\Pi - \Theta_k = 2\Pi - \Theta_i - \Theta_j \leq 2(\Pi - \Theta_i)$. Thus, we have $t < 2(\Pi - \Theta_i)$, which by Property 7.1, implies $Z(t,\Gamma_i) = 0$. Therefore, by (7.1), $Z(t,\Gamma_i) + Z(t,\Gamma_j) = Z(t,\Gamma_j) \leq t = t + Z(t,\Gamma_k)$.

Next, we focus on the remaining possibility in Case 3, namely, $0 \leq t'_{\Gamma_k} < t'_{\Gamma_i} \leq t'_{\Gamma_j}$. Applying (7.16) to $Z(t,\Gamma_i)$, $Z(t,\Gamma_j)$, and $Z(t,\Gamma_k)$, respectively, we have the following.

$$\begin{aligned}
Z(t,\Gamma_i) &= f(t'_{\Gamma_i},\Gamma_i), \\
&= \{\text{by (7.15) and } \Pi_i = \Pi\} \\
&\quad \left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor \cdot \Theta_i + \left(t'_{\Gamma_i} - \Pi \left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor - (\Pi - \Theta_i), 0 \right) \\
Z(t,\Gamma_j) &= f(t'_{\Gamma_j},\Gamma_j) \\
&= \{\text{rearranging}\} \\
&\quad f(t'_{\Gamma_i} + (t'_{\Gamma_j} - t'_{\Gamma_i}),\Gamma_j) \\
&\leq \{\text{by Property 7.5; note that } t'_{\Gamma_j} - t'_{\Gamma_i} \geq 0\} \\
&\quad f(t'_{\Gamma_i},\Gamma_j) + (t'_{\Gamma_j} - t'_{\Gamma_i}) \\
&= \{\text{by (7.2) and } \Pi_j = \Pi_i = \Pi\} \\
&\quad f(t'_{\Gamma_i},\Gamma_j) + \Theta_j - \Theta_i \\
&= \{\text{by (7.15) and } \Pi_j = \Pi\}
\end{aligned} \tag{7.30}$$

$$\left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor \cdot \Theta_j + \Theta_j - \Theta_i + \max\left(t'_{\Gamma_i} - \Pi \left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor - (\Pi - \Theta_j), 0 \right) \tag{7.31}$$

$$\begin{aligned}
Z(t,\Gamma_k) &= f(t'_{\Gamma_k}, \Gamma_j) \\
&= \{\text{rearranging}\} \\
&\quad f(t'_{\Gamma_i} - (t'_{\Gamma_i} - t'_{\Gamma_k}), \Gamma_k) \\
&\geq \{\text{by Property 7.5; note that } t'_{\Gamma_i} - t'_{\Gamma_k} \geq 0\} \\
&\quad f(t'_{\Gamma_i}, \Gamma_k) + (t'_{\Gamma_i} - t'_{\Gamma_k}) \\
&= \{\text{by (7.2) and } \Pi_i = \Pi_k = \Pi\} \\
&\quad f(t'_{\Gamma_i}, \Gamma_k) + \Theta_i - \Theta_k \\
&= \{\text{by (7.28)}\} \\
&\quad f(t'_{\Gamma_i}, \Gamma_k) + \Pi - \Theta_j \\
&= \{\text{by (7.15) and } \Pi_k = \Pi\} \\
&\quad \left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor \cdot \Theta_k + \Pi - \Theta_j + \\
&\qquad \max\left(t'_{\Gamma_i} - \Pi \left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor - (\Pi - \Theta_k), 0 \right)
\end{aligned} \tag{7.32}$$

For notational simplicity, we introduce the two terms below.

$$\Phi' = t'_{\Gamma_i} - \Pi \left\lfloor t'_{\Gamma_i} / \Pi \right\rfloor \tag{7.33}$$

$$\begin{aligned}
\Delta' = &\max\left(\Phi' - (\Pi - \Theta_i), 0\right) + \max\left(\Phi' - (\Pi - \Theta_j), 0\right) \\
&- \max\left(\Phi' - (\Pi - \Theta_k), 0\right)
\end{aligned} \tag{7.34}$$

Now, by (7.30), (7.31) and (7.32), we have

$$\begin{aligned}
&(Z(t,\Gamma_i) + Z(t,\Gamma_j)) - (t + Z(t,\Gamma_k)) \\
\leq &\left\lfloor \frac{t'_{\Gamma_i}}{\Pi} \right\rfloor (\Theta_i + \Theta_j - \Theta_k) + \Theta_j - \Theta_i - \Pi + \Theta_j - t + \Delta'
\end{aligned}$$

$$
\begin{aligned}
&= \{\text{rearranging and by (7.2) and (7.28)}\} \\
&\quad \left\lfloor t'_{\Gamma_i}/\Pi \right\rfloor \cdot \Pi + 2\Theta_j - \Theta_i - \Pi + \Delta' - (t'_{\Gamma_i} + \Pi - \Theta_i) \\
&= \{\text{rearranging}\} \\
&\quad \left\lfloor t'_{\Gamma_i}/\Pi \right\rfloor \cdot \Pi - t'_{\Gamma_i} - 2(\Pi - \Theta_j) + \Delta' \\
&= \{\text{by (7.33)}\} \\
&\quad \Delta' - \Phi' - 2(\Pi - \Theta_j). \qquad (7.35)
\end{aligned}
$$

Given the derivation above, we can complete the proof by showing $\Delta' - \Phi' - 2(\Pi - \Theta_j) \leq 0$. This result is implied by the following claim.

Claim 7.4. *In Case 3,* $\Delta' - \Phi' - 2(\Pi - \Theta_j) < 0$.

Proof. By (7.33), $0 \leq \Phi' < \Pi$. Also, by (7.29), $\Pi - \Theta_j \leq \Pi - \Theta_i < \Pi - \Theta_k$. Given these ranges, the following cases are exhaustive.

Case 3.1: $\Phi' \in [0, \Pi - \Theta_j)$, which implies $\Delta' - \Phi' - 2(\Pi - \Theta_j) = -\Phi' - 2(\Pi - \Theta_j) < 0$.

Case 3.2: $\Phi' \in [\Pi - \Theta_j, \Pi - \Theta_i)$, which implies $\Delta' - \Phi' - 2(\Pi - \Theta_j) = -3(\Pi - \Theta_j) < 0$.

Case 3.3: $\Phi' \in [\Pi - \Theta_i, \Pi - \Theta_k)$, which implies

$$
\begin{aligned}
&\quad \Delta' - \Phi' - 2(\Pi - \Theta_j) \\
&= \{\text{by (7.34)}\} \\
&\quad \Phi' - 3(\Pi - \Theta_j) - (\Pi - \Theta_i) \\
&< \{\text{in this case, } \Phi' < \Pi - \Theta_k \text{ holds}\} \\
&\quad (\Pi - \Theta_k) - 3(\Pi - \Theta_j) - (\Pi - \Theta_i) \\
&= \{\text{rearranging}\} \\
&\quad -2\Pi + 2\Theta_j + (\Theta_i + \Theta_j - \Pi - \Theta_k) \\
&= \{\text{by (7.28)}\} \\
&\quad -2\Pi + 2\Theta_j
\end{aligned}
$$

$$
\begin{aligned}
&< \{w_j < 1 \text{ in the lemma statement implies } \Theta_j < \Pi\} \\
&\quad 0.
\end{aligned}
$$

Case 3.4: $\Phi' \in [\Pi - \Theta_k, \Pi)$, which implies

$$
\begin{aligned}
&\Delta' - \Phi' - 2(\Pi - \Theta_j) \\
&= \{\text{by (7.34)}\} \\
&\quad -3\Pi + \Theta_i + 3\Theta_j - \Theta_k \\
&= \{\text{rearranging}\} \\
&\quad -2\Pi + 2\Theta_j + (\Theta_i + \Theta_j - \Pi - \Theta_k) \\
&= \{\text{by (7.28)}\} \\
&\quad -2\Pi + 2\Theta_j \\
&< \{w_j < 1 \text{ in the lemma statement implies } \Theta_j < \Pi\} \\
&\quad 0.
\end{aligned}
$$

□

Claim 7.4 and (7.35) together imply $Z(t,\Gamma_i) + Z(t,\Gamma_j) \leq t + Z(t,\Gamma_k)$, as required. □

Based on Lemma 7.1, we prove the following theorem by induction.

Theorem 7.2. *Given an arbitrary component $\mathcal{C} = (p, \mathcal{T})$ such that $(\forall \Gamma_i \in \mathcal{T} :: \Pi_i = \Pi)$, $\mathcal{C}$ is dominated by the MP-form component $\mathcal{C}' = (p^*, \mathcal{T}^*)$ such that $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$ and $(\forall \Gamma_i \in \mathcal{T}^* :: \Pi_i = \Pi)$.*

Proof. We prove the theorem by induction on $|\mathcal{T}|$.

Base Case: $|\mathcal{T}| \leq 1$**.** In this case, $\mathcal{C}$ and $\mathcal{C}^*$ are identical, because $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$ and $(\forall \Gamma_i \in \mathcal{T}^* :: \Pi_i = \Pi)$. Therefore, by Definition 7.3, $\mathcal{C}^*$ dominates $\mathcal{C}$.

Inductive Step. Suppose the theorem holds for any component C such that $|\mathcal{T}| \leq k$ where $k \geq 1$. We prove that it also holds for any component C such that $|\mathcal{T}| = k + 1$.

Because $k \geq 1$, $|\mathcal{T}| = k+1 \geq 2$. Therefore, $\mathcal{T}$ has at least two VPs that are not dedicated processors. Let Γ_i and Γ_j be two arbitrary such VPs. Without loss of generality, assume $0 < w_i \leq w_j < 1$.

To complete the proof, we show the existence of a component $\mathcal{C}' = (p', \mathcal{T}')$ such that $\mathcal{C}'$ has the same bandwidth and period as $\mathcal{C}$, but fewer VPs that are not dedicated processors, and $\mathsf{psf}_\infty(t, \mathcal{C}) \leq \mathsf{psf}_\infty(t, \mathcal{C}')$. $\mathcal{C}'$ is constructed via three cases that hinge on the value of $w_i + w_j$.

Case 1: If $0 < w_i + w_j < 1$, then let $p' = p$ and $\mathcal{T}' = \mathcal{T} \setminus \{\Gamma_i, \Gamma_j\} \cup \{\Gamma'_k\}$ where Γ'_k is a new VP such that $\Pi'_k = \Pi$ and $\Theta'_k = \Theta_i + \Theta_j$. Clearly, $\mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}')$. Also,

$$
\begin{aligned}
& \mathsf{psf}_\infty(t, \mathcal{C}) - \mathsf{psf}_\infty(t, \mathcal{C}') \\
= & \{\text{by (7.17)}\} \\
& (p - p')t + \sum_{\Gamma_l \in \mathcal{T}} Z(t, \Gamma_l) - \sum_{\Gamma_l \in \mathcal{T}'} Z(t, \Gamma_l) \\
= & Z(t, \Gamma_i) + Z(t, \Gamma_j) - Z(t, \Gamma'_k) \\
\leq & \{\text{by Lemma 7.1}\} \\
& 0.
\end{aligned}
$$

Case 2: If $w_i + w_j = 1$, then let $p' = p + 1$ and $\mathcal{T}' = \mathcal{T} \setminus \{\Gamma_i, \Gamma_j\}$. Clearly, $\mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}')$. Also,

$$
\begin{aligned}
& \mathsf{psf}_\infty(t, \mathcal{C}) - \mathsf{psf}_\infty(t, \mathcal{C}') \\
= & \{\text{by (7.17)}\} \\
& (p - p')t + \sum_{\Gamma_l \in \mathcal{T}} Z(t, \Gamma_l) - \sum_{\Gamma_l \in \mathcal{T}'} Z(t, \Gamma_l) \\
= & -t + Z(t, \Gamma_i) + Z(t, \Gamma_j) \\
\leq & \{\text{by Lemma 7.1}\} \\
& 0.
\end{aligned}
$$

Case 3: If $1 < w_i + w_j < 2$, then let $p' = p + 1$ and $\mathcal{T}' = \mathcal{T} \setminus \{\Gamma_i, \Gamma_j\} \cup \{\Gamma_k\}$ where Γ_k is a new VP such that $\Pi_k = \Pi$ and $\Theta_k = \Theta_i + \Theta_j - \Pi$. Clearly, $\mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}')$. Also,

$$
\mathsf{psf}_\infty(t, \mathcal{C}) - \mathsf{psf}_\infty(t, \mathcal{C}')
$$

$$
\begin{aligned}
&= \{\text{by (7.17)}\} \\
&\quad (p - p')t + \sum_{\Gamma_l \in \mathcal{T}} Z(t, \Gamma_l) - \sum_{\Gamma_l \in \mathcal{T}'} Z(t, \Gamma_l) \\
&= -t + Z(t, \Gamma_i) + Z(t, \Gamma_j) - Z(t, \Gamma'_k) \\
&\leq \{\text{by Lemma 7.1}\} \\
&\quad 0.
\end{aligned}
$$

In all three cases, the following two expressions hold.

$$\mathsf{bw}(\mathcal{C}') = \mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}^*) \tag{7.36}$$

$$\mathsf{psf}_\infty(t, \mathcal{C}) \leq \mathsf{psf}_\infty(t, \mathcal{C}') \tag{7.37}$$

Also, in Cases 1 and 3, we have $|\mathcal{T}'| = |\mathcal{T}| - 1$, while in Case 2, we have $|\mathcal{T}'| = |\mathcal{T}| - 2$, so $|\mathcal{T}'| \leq |\mathcal{T}| - 1 = (k+1) - 1 = k$. Therefore, by (7.36) and by the inductive hypothesis, $\mathcal{C}'$ is dominated by $\mathcal{C}^*$. Hence, by Definition 7.3,

$$\mathsf{psf}_\infty(t, \mathcal{C}') \leq \mathsf{psf}_\infty(t, \mathcal{C}^*). \tag{7.38}$$

By (7.37) and (7.38), $\mathsf{psf}_\infty(t, \mathcal{C}) \leq \mathsf{psf}_\infty(t, \mathcal{C}^*)$. Also, since $\mathcal{C}^*$ is in MP form, by Theorem 7.1, $\mathcal{C}^*$ dominates $\mathcal{C}$. □

The above theorem shows that, given a bandwidth and a common period shared by a set of asynchronous VPs, a component's supply is maximized when it is in MP form.

7.3.2 Different Periods

We now shift our focus by considering components that consist of a set of asynchronous VPs that may have different periods. Specifically, we consider a component $\mathcal{C} = (p, \mathcal{T})$, where for any two VPs Γ_i, Γ_j in $\mathcal{T}$, $\Pi_i \neq \Pi_j$ may hold. We investigate whether such a component $\mathcal{C}$ is dominated by a component in MP with the same bandwidth.

Towards this end, let $\mathcal{C}^*$ be a component in MP form such that $\mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}^*)$. To begin, note that if $\mathsf{bw}(\mathcal{C})$ is an integer, then $\mathcal{C}^*$ clearly dominates $\mathcal{C}$, because $\mathcal{C}^*$ has only dedicated processors that provide

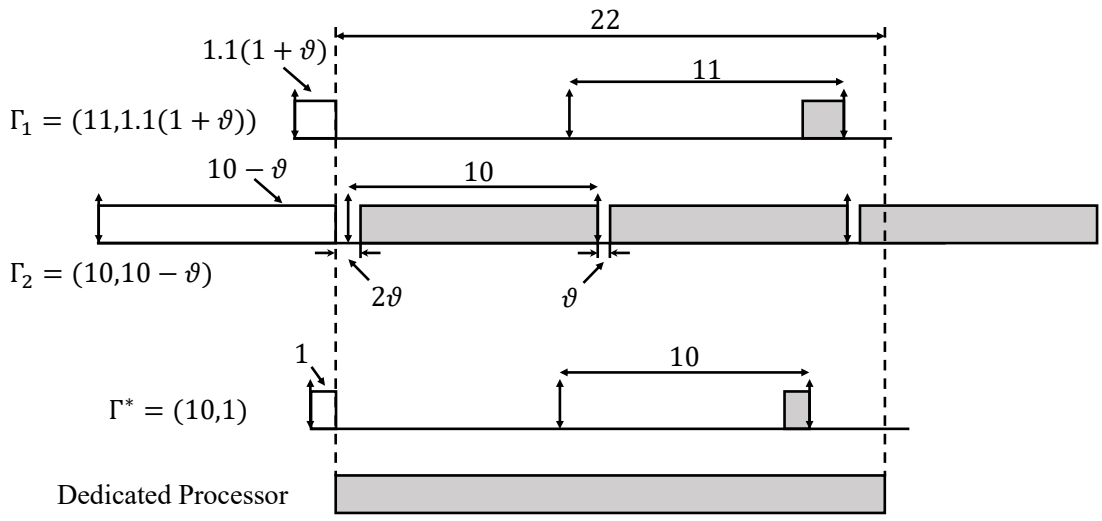


Figure 7.7: Illustration of the counterexample in Section 7.3.

supply constantly. In the rest of this section, we consider the more interesting case wherein $\mathsf{bw}(\mathcal{C})$ is not an integer. In this case, because $\mathcal{C}^*$ is in MP form, $|\mathcal{T}^*| = 1$. Let $\Gamma^* = (\Pi^*, \Theta^*)$ denote the lone VP in $\mathcal{T}^*$.

It is easy to see that, if $\mathcal{C}^*$ is to dominate $\mathcal{C}$, then the period Π^* generally will be dependent on the periods of the VPs in $\mathcal{C}$. In particular, if Π^* is selected to be very large in comparison to the periods of the VPs in $\mathcal{C}$, then Γ^* may be unable to gurantee any supply over relatively long intervals in which the VPs in $\mathcal{C}$ do. One obvious conjecture is that $\mathcal{C}^*$ will dominate $\mathcal{C}$ as long as $\Pi^* \leq \min\{\Pi_i \,|\, \Gamma_i \in \mathcal{T}\}$ holds. However, the following counterexample shows that this conjecture is not true.

Counterexample. Consider a component $\mathcal{C}$ with these two VPs: $\Gamma_1 = (11, 1.1(1+\vartheta))$ and $\Gamma_2 = (10, 10-\vartheta)$, where ϑ is an arbitrary small positive real number, *i.e.*, $\vartheta \to 0^+$. An MP-form component $\mathcal{C}^*$ with the same bandwidth also has two VPs: a dedicated processor and $\Gamma^* = (10,1)$. Note that, in this setting, $\Pi^* \leq \min\{\Pi_i \,|\, \Gamma_i \in \mathcal{T}\}$ holds. As illustrated in Figure 7.7, $\mathsf{psf}_\infty(22,\mathcal{C}) = 1.1(1+\vartheta)+22-4\vartheta = 23.1-2.9\vartheta$, while $\mathsf{psf}_\infty(22,\mathcal{C}^*) = 1+22 = 23$. Because $\vartheta \to 0^+$, $23.1-2.9\vartheta > 23$. That is, $\mathsf{psf}_\infty(22,\mathcal{C}) > \mathsf{psf}_\infty(22,\mathcal{C}^*)$, which implies that $\mathcal{C}^*$ does not dominate $\mathcal{C}$.

Despite the negative implications of this counterexample, we show next that $\mathcal{C}^*$ does indeed dominate $\mathcal{C}$ if Π^* is further restricted.

Theorem 7.3. *$\mathcal{C}$ is dominated by the MP-form component $\mathcal{C}^*$ as defined above as long as $\Pi^* \leq \frac{1}{2}\min\{\Pi_i \,|\, \Gamma_i \in \mathcal{T}\}$.*

Proof. Given $\mathcal{C}$, we first construct a new component $\mathcal{C}'$ such that $p' = p$ and $|\mathcal{T}'| = |\mathcal{T}|$. Each $\Gamma'_i = (\Pi'_i, \Theta'_i) \in \mathcal{T}'$ is constructed from the VP $\Gamma_i = (\Pi_i, \Theta_i) \in \mathcal{T}$ by defining $\Pi'_i = \Pi^*$ and $\Theta'_i = \Theta_i \frac{\Pi^*}{\Pi_i}$. These definitions imply

$$w'_i = \frac{\Theta'_i}{\Pi'_i} = \frac{\Theta_i}{\Pi_i} = w_i. \tag{7.39}$$

By Property 7.2,

$$\begin{aligned} Z(t, \Gamma'_i) \geq & \max\{(t - 2(\Pi'_i - \Theta'_i))w'_i, 0\} \\ = & \{\text{by (7.39) and because } \Pi'_i = \Pi^*\} \\ & \max\{(t - 2\Pi^*(1 - w_i))w_i, 0\} \\ \geq & \{\text{because } \Pi^* \leq \frac{1}{2}\min\{\Pi_i \mid \Gamma_i \in \mathcal{T}\}\} \\ & \max\{(t - \Pi_i(1 - w_i))w_i, 0\}. \end{aligned}$$

On the other hand, by Property 7.3,

$$\begin{aligned} Z(t, \Gamma_i) \leq & \max\{(t - (\Pi_i - \Theta_i))w_i, 0\} \\ = & \max\{(t - \Pi_i(1 - w_i))w_i, 0\}, \end{aligned}$$

from which we can conclude the following.

$$(\forall i : 1 \leq i \leq |\mathcal{T}| = |\mathcal{T}'| :: Z(t, \Gamma_i) \leq Z(t, \Gamma'_i)) \tag{7.40}$$

Also, $p = p'$, and therefore, by (7.17)

$$\mathsf{psf}_\infty(t, \mathcal{C}) \leq \mathsf{psf}_\infty(t, \mathcal{C}'). \tag{7.41}$$

Because $(\forall \Gamma'_i :: \Pi'_i = \Pi^*)$ holds, and by (7.39), $\mathsf{bw}(\mathcal{C}') = p' + \sum_{\Gamma'_i \in \mathcal{T}'} w'_i = p + \sum_{\Gamma_i \in \mathcal{T}} w_i = \mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}^*)$ holds, $\mathcal{C}'$ is a component in which all VPs share the same period, and its bandwidth equals the bandwidth of the MP-form component $\mathcal{C}^*$. Therefore, by Theorem 7.2, $\mathcal{C}^*$ dominates $\mathcal{C}'$. By Definition 7.3, this implies

$$\mathsf{psf}_\infty(t, \mathcal{C}') \leq \mathsf{psf}_\infty(t, \mathcal{C}^*). \tag{7.42}$$

By (7.41) and (7.42), $\mathsf{psf}_\infty(t,\mathcal{C}) \leq \mathsf{psf}_\infty(t,\mathcal{C}^*)$ holds, so by Theorem 7.1, the MP-form component $\mathcal{C}^*$ dominates $\mathcal{C}$. □

In some cases, the dominance of $\mathcal{C}^*$ over $\mathcal{C}$ can be established with a weaker restriction on the period Π^*. The following theorem gives such a case; note that harmonic and loose-harmonic[2] periods satisfy the condition given in this theorem.

Theorem 7.4. *For the component $\mathcal{C} = (p,\mathcal{T})$ defined above, let $\Pi_{min} = \min\{\Pi_i \mid \Gamma_i \in \mathcal{T}\}$. If the condition $(\forall \Gamma_i \in \mathcal{C} :: \Pi_i = \Pi_{min} \vee \Pi_i \geq 2\Pi_{min})$ holds, then $\mathcal{C}$ is dominated by the MP-form component $\mathcal{C}^*$ as defined above if Π^* is set equal to Π_{min}.*

Proof. We construct $\mathcal{C}'$ in the same way as in the proof of Theorem 7.3 such that $\Pi'_i = \Pi^* = \Pi_{min}$ and $\Theta'_i = \Theta_i \frac{\Pi^*}{\Pi_i}$. Given the statement of Theorem 7.4, we have for each i, $\Pi_i = \Pi^* = \Pi'_i$ or $\Pi_i \geq 2\Pi_{min} = 2\Pi^* = 2\Pi'_i$. In the former case, $Z(t,\Gamma_i) = Z(t,\Gamma'_i)$ holds; in the latter case, $Z(t,\Gamma_i) \leq Z(t,\Gamma'_i)$ can be shown to follow from Properties 7.2 and 7.3 using the same reasoning as in the proof of Theorem 7.3. Thus, we can establish (7.40) in the context of this new theorem, and then show exactly as done in the proof of Theorem 7.3 that $\mathcal{C}^*$ dominates $\mathcal{C}$. □

7.4 Synchronous and Concrete Asynchronous

In this section, we consider components consisting of VPs with specified phases, *i.e.*, both *concrete asynchronous* and *synchronous* VPs. These VPs may have either a common period or different periods.

The case of synchronous VPs and a common period is highly related to the MPR model (Shin and Lee, 2003), as that model enforces both of these requirements. The following theorem is easily implied by prior work on the MPR model (Xu et al., 2015) that shows that, by enforcing MP form, a component abstracted by the MPR model achieves its maximum supply.

Theorem 7.5. *(Follows from (Xu et al., 2015)) If $\mathcal{C} = (p,\mathcal{T})$ is a synchronous component and $(\forall \Gamma_i \in \mathcal{T} :: \Pi_i = \Pi)$ holds, then it is dominated by the MP-form component $\mathcal{C}' = (p^*,\mathcal{T}^*)$, where $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$ and $(\forall \Gamma_i \in \mathcal{T}^* :: \Pi_i = \Pi)$.*

Because synchronous VPs are a special case of concrete asynchronous VPs where all VP phases happen to be zero, one might expect that Theorem 7.5 can be extended to concrete asynchronous VPs, and speculate

[2]The smallest period divides any larger period.

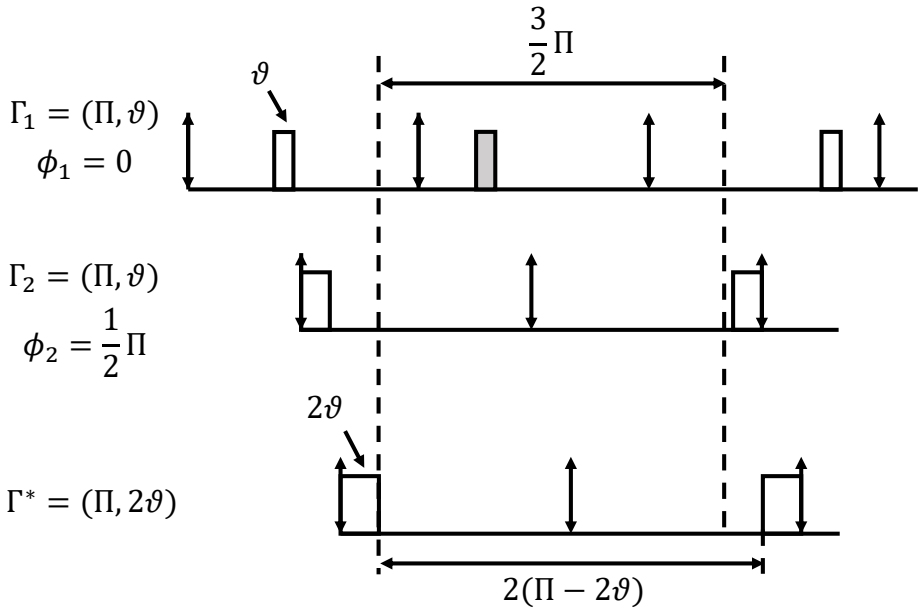


Figure 7.8: Illustration of the counterexample in Section 7.4.

that an arbitrary concrete asynchronous component with a common period is dominated by the MP-form component of the same bandwidth and period. However, this is unfortunately not true.

Counterexample. Consider a non-MP-form component $\mathcal{C}$ that has two VPs, $\Gamma_1 = (\Pi, \vartheta)$ and $\Gamma_2 = (\Pi, \vartheta)$, with a common period and arbitrarily small budget, *i.e.*, $\vartheta \to 0^+$. Suppose these two VPs have different phases, $\phi_1 = 0$ and $\phi_2 = \frac{1}{2}\Pi$, as shown in Figure 7.8. Observe that any time interval of length $\frac{3}{2}\Pi$ must include exactly one period of allocation of Γ_1 or Γ_2. Therefore, $\mathsf{psf}_1(\frac{3}{2}\Pi, \mathcal{C}) \geq \vartheta$. In contrast, consider the MP-form counterpart of $\mathcal{C}$: $\mathcal{C}^* = \{\Gamma^*\}$, where $\Gamma^* = (\Pi, 2\vartheta)$. By the worst case illustrated in Figure 7.8, $\mathsf{psf}_1(t, \mathcal{C}^*) = 0$ holds for any $t \leq 2(\Pi - 2\vartheta)$. Because $\vartheta \to 0^+$, we have $\frac{3}{2}\Pi < 2(\Pi - 2\vartheta)$. Therefore, $\mathsf{psf}_1(\frac{3}{2}\Pi, \mathcal{C}^*) = 0 < \vartheta \leq \mathsf{psf}_1(\frac{3}{2}\Pi, \mathcal{C})$, which implies that $\mathcal{C}^*$ does not dominate $\mathcal{C}$.

Nonetheless, we provide next a theorem that shows that a component in non-MP-form will still be dominated by an MP-form component of the same bandwidth, provided the period of the latter is properly selected. Furthermore, the required period selection is valid not only for concrete asynchronous VPs with a common period, but also for synchronous or concrete asynchronous VPs with different periods. The theorem is stated assuming concrete asynchronous VPs, a category that subsumes these other possibilities.

Theorem 7.6. *If $\mathcal{C} = (p, \mathcal{T})$ is a concrete asynchronous component, then it is dominated by the MP-form component $\mathcal{C}' = (p^*, \mathcal{T}^*)$, where $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$, provided the following condition holds: if $|\mathcal{T}^*| = 1$, then*

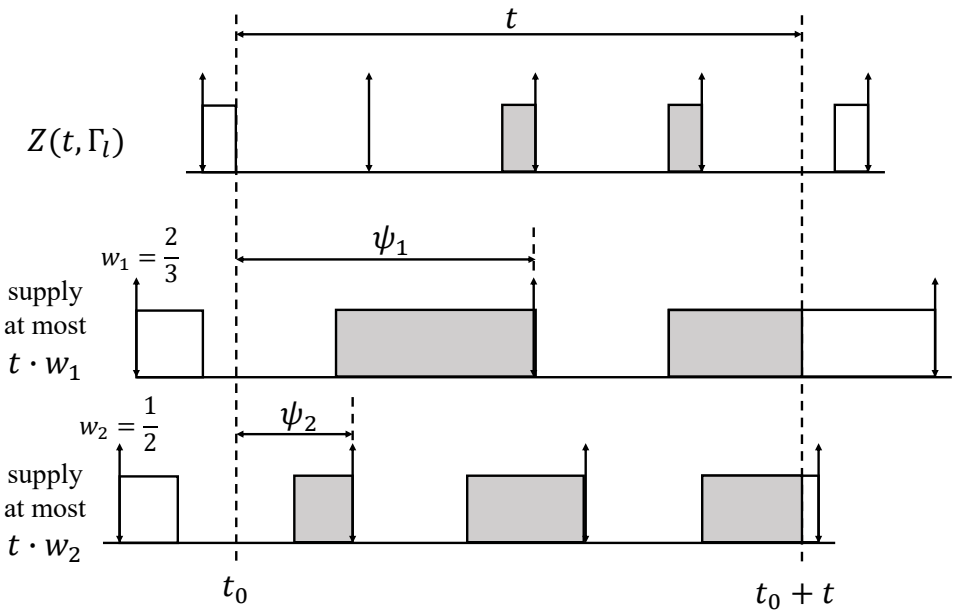


Figure 7.9: A possible scenario for any concrete phases.

the period of the lone VP in $\mathcal{T}$ must satisfy

$$\Pi^* \leq \frac{\Pi_l(1-w_l)w_l}{2(1-w^*)w^*}, \tag{7.43}$$

where l is defined by

$$\Pi_l - \Theta_l = \min\{\Pi_i - \Theta_i \mid \Gamma_i \in \mathcal{T}\}. \tag{7.44}$$

Proof. Because $\mathcal{C}^*$ is in MP form, $|\mathcal{T}| \leq 1$ holds. If $|\mathcal{T}| = 0$ holds, then $\mathcal{C}^*$ has dedicated processors only. Because $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$ is assumed, this clearly implies that $\mathcal{C}^*$ dominates $\mathcal{C}$. In the rest of the proof, we focus on the more interesting case wherein $|\mathcal{T}^*| = 1$ holds. In this case, $\mathsf{bw}(\mathcal{C}^*)$ is not integral, so $\mathsf{bw}(\mathcal{C})$ is also not integral. This implies that $|\mathcal{T}| > 0$ holds. We now show that $\mathsf{psf}_\infty(t,\mathcal{C}) \leq \mathsf{psf}_\infty(t,\mathcal{C}^*)$ holds by considering two cases.

Case 1: $t \leq \Pi_l - \Theta_l$**.** By Claim 7.1 and (7.44), any VP Γ_i in $\mathcal{T}$ can provide zero supply within any time interval of length t, where $t \leq \Pi_l - \Theta_l$. Within any such time interval, the p dedicated processors of $\mathcal{C}$ provide supply continually. Because $\mathcal{C}^*$ is in MP form, $p \leq p^*$ holds. Therefore, $\mathsf{psf}_\infty(t,\mathcal{C}) = t \cdot p \leq t \cdot p^* \leq \mathsf{psf}_\infty(t,\mathcal{C}^*)$.

Case 2: $t > \Pi_l - \Theta_l$**.** Let Γ_l be a VP such that $\Pi_l - \Theta_l = \min\{\Pi_i - \Theta_i \mid \Gamma_i \in \mathcal{T}\}$. Then, the allocations described next and illustrated in Figure 7.9 are *possible* for any concrete VP phases (*i.e.*, synchronous or

concrete asynchronous). Let t_0 be a time instant such that Γ_l gets its minimal supply $Z(t,\Gamma_l)$ within the time interval $[t_0, t_0+t)$. For any other VP Γ_j, where $j \neq l$, let ψ_j denote the distance from t_0 to the start of its next allocation period, *i.e.*, the next allocation period of Γ_j at or after time t_0 starts at time $t_0+\psi_j$. (Note that the value of ψ_j will depend on the phase of Γ_j.) In this *possible* allocation sequence, if Γ_j has an allocation period that includes t_0 (as depicted), then assume that it provides a supply of $(\Pi_j - \psi_j)\cdot w_j$ time units within that allocation period before t_0, *i.e.*, in $[t_0-(\Pi_j-\psi_j), t_0)$. Regardless of whether Γ_j has an allocation period that includes t_0, assume that it provides supply as late as possible in each of its allocation periods beyond time t_0. It is easy to show that, in this situation, each Γ_j provides a supply of at most $t\cdot w_j$ time units during $[t_0, t_0+t)$. By Definition 7.2, the PSF functions capture the *minimum* allocation that can occur, which is upper bounded by that demonstrated in the *possible* allocation sequence just discussed. Therefore, we have

$$
\begin{aligned}
&\mathsf{psf}_\infty(t,\mathcal{C}) \\
\leq\ & t\cdot p + Z(t,\Gamma_l) + \sum_{\Gamma_j\in\mathcal{T}\wedge j\neq l} t\cdot w_j \\
\leq\ & \{\text{by Property 7.3}\} \\
& t\cdot p + \max\{w_l\cdot(t-(\Pi_l-\Theta_l)),0\} + \sum_{\Gamma_j\in\mathcal{T}\wedge j\neq l} t\cdot w_j. \\
\leq\ & \{\text{by our assumption in Case 2 that } t > \Pi_l-\Theta_l \text{ holds}\} \\
& t\cdot p + w_l\cdot(t-(\Pi_l-\Theta_l)) + \sum_{\Gamma_j\in\mathcal{T}\wedge j\neq l} t\cdot w_j. \\
=\ & \{\text{rearranging}\} \\
& t\cdot\left(p+\sum_{\Gamma_i\in\mathcal{T}} w_i\right) - w_l\cdot(\Pi_l-\Theta_l) \\
=\ & \{\text{by (7.5) and the definition of } w_l\} \\
& t\cdot \mathsf{bw}(\mathcal{C}) - \Pi_l(1-w_l)w_l. \qquad (7.45)
\end{aligned}
$$

By Claim 7.2 and our assumption that $|\mathcal{T}^*| = 1$ holds, we have

$$
\begin{aligned}
&\mathsf{psf}_\infty(t,\mathcal{C}^*) \\
=\ & t\cdot p^* + Z(t,\Gamma^*) \\
\geq\ & \{\text{by Property 7.2}\}
\end{aligned}
$$

$$
\begin{aligned}
& t \cdot p^* + \max\{w^* \cdot (t - 2(\Pi^* - \Theta^*)), 0\} \\
\geq\ & \{\text{because } \max\{x, y\} \geq x\} \\
& t \cdot p^* + w^* \cdot (t - 2(\Pi^* - \Theta^*)) \\
=\ & \{\text{rearranging and using the definition of } w^*\} \\
& t \cdot (p^* + w^*) - 2\Pi^*(1 - w^*)w^* \\
=\ & \{\text{by (7.5)}\} \\
& t \cdot \mathsf{bw}(\mathcal{C}^*) - 2\Pi^*(1 - w^*)w^*. \qquad (7.46)
\end{aligned}
$$

By (7.45) and (7.46),

$$
\begin{aligned}
& \mathsf{psf}_\infty(t, \mathcal{C}) - \mathsf{psf}_\infty(t, \mathcal{C}^*) \\
\leq\ & \{\text{because } \mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}^*)\} \\
& 2\Pi^*(1 - w^*)w^* - \Pi_l(1 - w_l)w_l \\
\leq\ & \{\text{by (7.43)}\} \\
& 0.
\end{aligned}
$$

That is, $\mathsf{psf}_\infty(t, \mathcal{C}) \leq \mathsf{psf}_\infty(t, \mathcal{C}^*)$ for $t > \Pi_l - \Theta_l$.

Combining Cases 1 and 2, we have $\mathsf{psf}_\infty(t, \mathcal{C}) \leq \mathsf{psf}_\infty(t, \mathcal{C}^*)$ for any $t \geq 0$. Also, $\mathcal{C}^*$ is in MP form. Thus, by Theorem 7.1, $\mathcal{C}^*$ dominates $\mathcal{C}$. □

7.5 Indomitability of MP Form

Although we have shown that an arbitrary component can always be dominated by a component in MP form with the same bandwidth, this result requires restrictions on the period of the MP-form component in some cases. This raises the question of whether the dominance is really due to the definition of MP form or just side effect of the period restrictions. In this section, we address this question. We show that an MP-form component can never be dominated by a non-MP-form component of the same bandwidth, regardless of any restrictions that may be applied to the non-MP-form component.

The following theorem holds, regardless of whether the VPs are synchronous, concrete asynchronous, or non-concrete asynchronous.

Theorem 7.7. *Given an MP-form component $\mathcal{C}^*$ and an arbitrary non-MP-form component $\mathcal{C}$ such that $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$ holds, $\mathcal{C}$ does not dominate $\mathcal{C}^*$, no matter how $\{\Pi_i \mid \Gamma_i \in \mathcal{C}\}$ is defined.*

Proof. Let p and p^* denote the number of dedicated processors in $\mathcal{C}$ and $\mathcal{C}^*$, respectively. Because $\mathcal{C}^*$ is in MP form and $\mathsf{bw}(\mathcal{C}) = \mathsf{bw}(\mathcal{C}^*)$ holds, we have $p \leq p^*$. We consider the two cases $p < p^*$ and $p = p^*$ separately below.

Case 1: $p < p^*$**.** By Claim 7.1, regardless of the VPs' phases, the supply of each VP $\Gamma_i \in \mathcal{T}$ can be zero for *any* time interval of length t such that $0 < t \leq \Pi_i - \Theta_i$, so $\mathsf{psf}_\infty(t, \mathcal{C}) = t \cdot p$ for any t such that $0 < t \leq t_s$, where $t_s = \min\{\Pi_i - \Theta_i \,|\, \Gamma_i \in \mathcal{T}\}$. On the other hand, $\mathsf{psf}_\infty(t, \mathcal{C}^*) \geq t \cdot p^*$ for any $t > 0$. Thus, for any t such that $0 < t \leq t_s$, we have $\mathsf{psf}_\infty(t, \mathcal{C}) = t \cdot p < t \cdot p^* \leq \mathsf{psf}_\infty(t, \mathcal{C}^*)$, *i.e.*, $\mathsf{psf}_\infty(t, \mathcal{C}) < \mathsf{psf}_\infty(t, \mathcal{C}^*)$. Note that the stated range for t is not vacuous. This is because $\mathcal{C}$ is not in MP form, which implies that $|\mathcal{T}| > 0$ holds, and hence that $t_s > 0$ holds as well. Because $\mathsf{psf}_\infty(t, \mathcal{C}) < \mathsf{psf}_\infty(t, \mathcal{C}^*)$ holds, by Definition 7.3, $\mathcal{C}$ does not dominate $\mathcal{C}^*$.

Case 2: $p = p^*$**.** In this case, we have $|\mathcal{T}^*| = 1$, because if $|\mathcal{T}^*| = 0$ holds, then either $\mathcal{C}$ is also in MP form or $\mathsf{bw}(\mathcal{C}) > \mathsf{bw}(\mathcal{C}^*)$, neither of which is allowed by the statement of the theorem. Let Γ^* denote the lone VP in $\mathcal{C}^*$ and let w^* denote its bandwidth. Then, $w^* = \sum_{\Gamma_i \in \mathcal{T}} w_i$, since $\mathsf{bw}(\mathcal{C}^*) = \mathsf{bw}(\mathcal{C})$. Also, because $\mathcal{C}$ is not in MP form, by Definition 7.1, both $|\mathcal{T}| \geq 2$ and $(\forall \Gamma_i \in \mathcal{T} :: w_i > 0)$ hold. Therefore, $(\forall \Gamma_i \in \mathcal{T} :: w_i < w^*)$. Letting $w_{max} = \max\{w_i \mid \Gamma_i \in \mathcal{T}\}$, this implies

$$w_{max} < w^*. \tag{7.47}$$

Let δ be the greatest common divisor of the values in $\{\Pi_i \mid \Gamma_i \in \mathcal{T}\}$. Then, the processor-time allocation illustrated in Figure 7.10, where every VP provides $\delta \cdot w_i$ time units of processor time at the end of every *aligned* time window of δ time units, is possible regardless of any assumptions regarding the VPs' phases. This is because, in this schedule, each VP Γ_i is allocated Θ_i time units within *any* time interval of length Π_i. Such an allocation satisfies the specification of Γ_i regardless of how phases are defined. Under this allocation pattern, each VP other than the one with the maximum bandwidth w_{max} provides all of its supply in parallel with that maximum-bandwidth VP. Furthermore, with the depicted allocations, the minimum supply during

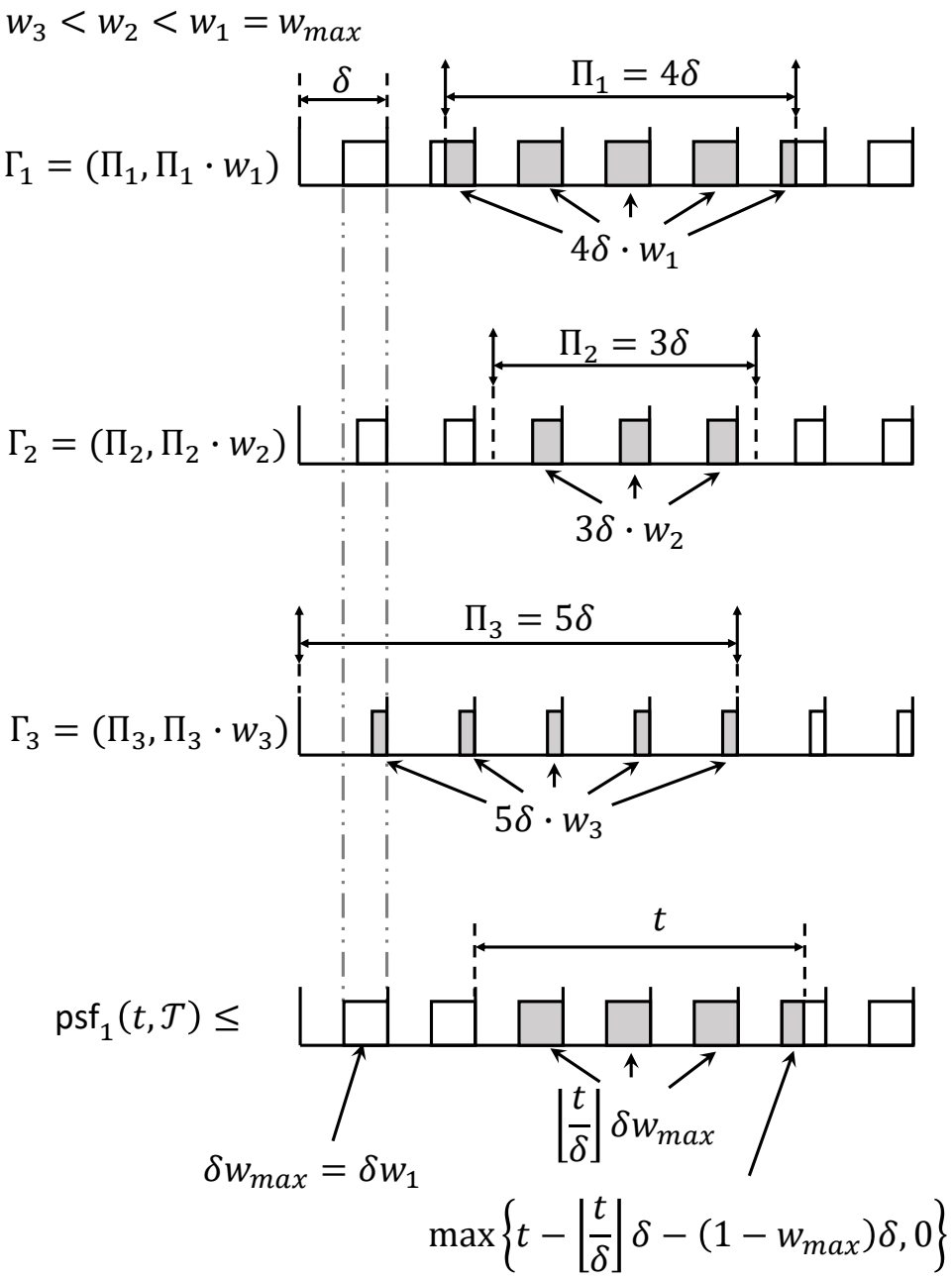


Figure 7.10: Illustration of Case 2 of Theorem 7.7.

any time interval of length t with a degree of parallelism of *one* is

$$\left\lfloor \frac{t}{\delta} \right\rfloor \delta w_{max} + \max\{t - \left\lfloor \frac{t}{\delta} \right\rfloor \delta - (1 - w_{max})\delta, 0\} \leq t \cdot w_{max}.$$

Because the PSF functions, by Definition 7.2, capture the worst case among all possible allocation scenarios,

$$\mathsf{psf}_1(t, \mathcal{T}) \leq t \cdot w_{max}. \tag{7.48}$$

Therefore, given that $\mathcal{C}$ has p dedicated processors,

$$\mathsf{psf}_{p+1}(t, \mathcal{C}) = tp + \mathsf{psf}_1(t, \mathcal{T}) \leq t(p + w_{max}). \tag{7.49}$$

On the other hand, for $\mathcal{C}^*$, for any $t \geq 2(\Pi^* - \Theta^*) = 2\Pi^*(1 - w^*)$, by (7.17)

$$\begin{aligned}
&\mathsf{psf}_{p^*+1}(t, \mathcal{C}^*) \\
&= \{\text{by (7.8) and because } \mathcal{C}^* \text{ is in MP form}\} \\
&\quad \mathsf{psf}_{\infty}(t, \mathcal{C}^*) \\
&= \{\text{by (7.17)}\} \\
&\quad tp^* + Z(t, \Gamma^*) \\
&\geq \{\text{by Property 7.2, and since } t \geq 2(\Pi^* - \Theta^*)\} \\
&\quad tp^* + w^*(t - 2(\Pi^* - \Theta^*)) \\
&= \{\text{rearranging and using } w^* = \Theta^*/\Pi^*\} \\
&\quad t(p^* + w^*) - 2\Pi^* w^*(1 - w^*).
\end{aligned}$$

Because $p = p^*$ holds in Case 2,

$$\mathsf{psf}_{p+1}(t, \mathcal{C}^*) \geq t(p + w^*) - 2\Pi^* w^*(1 - w^*). \tag{7.50}$$

By (7.49) and (7.50), for any $t \geq 2\Pi^*(1 - w^*)$,

$$\mathsf{psf}_{p+1}(t, \mathcal{C}^*) - \mathsf{psf}_{p+1}(t, \mathcal{C}) \geq t(w^* - w_{max}) - 2\Pi^* w^*(1 - w^*).$$

Hence, by (7.47), for any $t > \frac{2\Pi^* w^* (1-w^*)}{w^* - w_{max}} > 2\Pi^*(1-w^*)$, $\mathsf{psf}_{p+1}(t, \mathcal{C}) < \mathsf{psf}_{p+1}(t, \mathcal{C}^*)$.

Thus, by Definition 7.3, $\mathcal{C}$ does not dominate $\mathcal{C}^*$. Note that the above argument is valid regardless of the definition of $\{\Pi_i \mid \Gamma_i \in \mathcal{C}\}$. □

Theorem 7.7 shows that, no matter how the periods of a non-MP-form component are defined, it cannot dominate any component in MP form with the same total bandwidth.

7.6 Chapter Summary

In this chapter, we studied processor allocations to components comprised of multiple VPs, which may be synchronous, concrete asynchronous, or non-concrete asynchronous. We showed that any arbitrary component is always dominated by an MP-form component of the same bandwidth, provided the period used in defining the MP-form component meets certain requirements. We also showed that a component in MP form can never be dominated by any non-MP-form component of the same bandwidth, regardless of how periods are defined.

CHAPTER 8: CONCLUSION

Real-time scheduling analysis is crucial for time-critical systems, in which provable timing guarantees are more important than observed raw performance. Although significant work has been done with respect to real-time scheduling on symmetric multiprocessor platforms, the same is not true for asymmetric ones. The main objective of the research presented in this dissertation was to provide fundamental results and analysis techniques to mitigate this insufficiency in the literature. Towards this goal, we designed and analyzed a few real-time scheduling algorithms under several system models, addressing asymmetric multiprocessor platforms due to differing processor speeds, processor functionalities, or virtualization, respectively. In the following, we summarize the results presented in this dissertation (Section 8.1), briefly describe other publications by the author that have been done in parallel with but beyond the scope of this dissertation (Section 8.2), and discuss future work (Section 8.3).

8.1 Summary of Results

Focusing on real-time scheduling on asymmetric multiprocessor platforms, the results presented in this dissertation can be summarized as follows.

Results regarding the SRT-optimality of G-EDF on uniform multiprocessors. In Chapter 3, we answered the question: is G-EDF is SRT-optimal on uniform multiprocessors? The answer was different from that for the similar question with respect to identical multiprocessors, where both preemptive and non-preemptive G-EDF are SRT-optimal (Devi and Anderson, 2008). By providing a counterexample, we showed that non-preemptive G-EDF is not SRT-optimal for uniform multiprocessors. On the other hand, by developing a new framework to bound tardiness, we proved that preemptive G-EDF is indeed SRT-optimal for uniform multiprocessors. Both results in fact apply to a broader range of problems beyond the G-EDF scheduling of sporadic tasks. The negative result applies to any work-conserving non-preemptive scheduling algorithm, whereas the positive result applies the VPP task model, which is a more general task model than the sporadic task model.

Two semi-partitioned scheduling algorithms for uniform multiprocessors. In Chapter 4, we presented two semi-partitioned scheduling algorithms designed for uniform multiprocessors, namely EDF-sh and EDF-tu. Both algorithms limit the number of migrating tasks: for scheduling n tasks on m uniform processors, EDF-sh allows at most $m-1$ tasks to migrate while EDF-tu allows m. EDF-sh further requires these migrating tasks to migrate at job boundaries only, at the cost of supporting SRT tasks only and being not SRT-optimal. In contrast, EDF-tu is SRT-optimal and includes a tunable parameter, the frame size. For any positive value of the frame size, tardiness is guaranteed to be at most this value. Furthermore, if the frame size divides all task periods, EDF-sh becomes HRT-optimal and ensures zero tardiness.

Allowing intra-task parallelism on uniform multiprocessors. In Chapter 5, we introduced a new task model, called the npc-sporadic task model. In contrast to the conventional sporadic task model where jobs of the same task must execute in sequence, such jobs are allowed to execute in parallel in the npc-sporadic task model. For scheduling npc-sporadic tasks on a uniform multiprocessor, the HRT-feasibility condition is the same as that for the conventional sporadic task model; however, the SRT-feasibility condition for the npc-sporadic model merely requires that the system is not overutilized while the rather complicated per-task utilization constraint for the conventional sporadic task model can be eliminated. We further showed that both preemptive and non-preemptive G-EDF are SRT-optimal on uniform multiprocessors for scheduling npc-sporadic task systems. Preemptive G-EDF is more greedy in executing jobs on faster processors and hence has a better response-time bound, at the expense of potentially greater preemption and migration frequencies. On the other hand, non-preemptive G-EDF does not preempt or migrate jobs, but its guaranteed response-time bounds are relatively higher.

Techniques for deriving and improving end-to-end response-time bounds for DAG-based task systems on unrelated heterogeneous platforms. In Chapter 6, we addressed the problem of scheduling real-time dataflows on heterogeneous CEs. We formalized this problem by representing each dataflow by a DAG, the nodes (resp., edges) of which represent tasks (resp., producer/consumer relationships). We then presented task-transformation techniques to provide end-to-end response-time bounds for such DAG-based task systems implemented on heterogeneous multiprocessor platforms. We further presented an LP-based method for setting relative deadlines that can be applied to improve these bounds. In addition, the early-releasing technique was shown to be able to improve observed end-to-end response times while not compromising their analytical bounds. For systems where multiple DAGs are structurally identical, which is the case in our

targeted application domain, we also presented a DAG combining technique that was shown to be able to further improve the end-to-end response-time bounds for DAGs.

Dominance of MP-form supply on virtual multiprocessor platforms. In Chapter 7, we focused on VP allocation schemes for constituting a virtual multiprocessor platform, or a component, on a symmetric physical multiprocessor platform. Even for a designated component capacity and a given physical platform, there are an infinite number of VP allocation schemes to apply and a choice must be made. Furthermore, the allocation periods of VPs may be synchronous, concrete asynchronous, or non-concrete asynchronous, and the sizes of such periods may be the same for all VPs or may vary from one VP to another. In each of these cases, a VP allocation scheme, called MP form, is shown to dominate any other scheme. Under MP form, each component is allocated at most one partially available processor, with all other processors allocated to it being fully available. Specifically, we showed that any arbitrary component is always dominated by an MP-form component of the same bandwidth, provided the period used in defining the MP-form component meets certain requirements. We also showed that a component in MP form can never be dominated by any non-MP-form component of the same bandwidth, regardless of how periods are defined.

8.2 Other Publications

The following is a brief summary of other work done by the author during his doctoral studies.

Supporting real-time computer-vision workloads using OpenVX.[1] For computer-vision algorithms, graphics processing units (GPUs) are a particularly compelling accelerator to consider, as GPUs are well suited for efficiently performing the matrix-oriented computations inherent in many computer-vision applications. To ease the development of such applications on heterogeneous platforms such as those in which GPUs are employed, and to enable system-level optimization (Rainey et al., 2014), a standard, called OpenVX, has been created and ratified (Khronos Group, 2014). In OpenVX, a set of basic operations, or *primitives*,[2] commonly used in computer-vision algorithms are provided to programmers. OpenVX also defines a set

[1] Details of this contribution have been published in the following papers:

Elliott, G., Yang, K., and Anderson, J. (2015). Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*, pages 273–284.

Yang, K., Elliott, G., and Anderson, J. (2015). Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, pages 77–86.

[2] In OpenVX, these basic operations are called "kernels."

of *data objects*,[3] and has a graph-based execution model. The programmer constructs a computer-vision algorithm by instantiating primitives as *nodes* and data objects as *parameters* and binding parameters to node inputs and outputs. Node dependencies (*i.e.*, edges) are not explicitly declared. Rather, the structure of a graph is derived from how parameters are bound to nodes. A video frame is processed by executing such an OpenVX graph end-to-end. Since each node may use a mix of the processing elements of a heterogeneous platform, OpenVX enables a single computer-vision algorithm to execute across CPUs, GPUs, DSPs, *etc.*.

However, the original OpenVX specification does not fit in any existing real-time task model and therefore no real-time analysis framework can apply directly. In (Elliott et al., 2015; Yang et al., 2015), we modified an OpenVX modification by NVIDIA, called VisionWorks®, to enable modeling each node in an OpenVX graph as a sporadic task. We then showed that, for acyclic OpenVX graphs, real-time scheduling and analysis techniques for DAG-based systems (Liu and Anderson, 2010) can be applied to derive an end-to-end response-time bound for each DAG. However, in some OpenVX graphs, cycles may exist due to *delay edges*, which specify data dependencies on the processing of *prior* frames. To address this problem, we presented techniques to refine a cyclic OpenVX graph to be acyclic while preserving all precedence constraints among tasks. Finally, in contrast to the original OpenVX specification that requires a graph to execute end-to-end before it may be executed again, our modification enabled graph pipelining. To support this, we resolved a data-object overwriting problem by replicating data objects and we derived an upper bound on the number of needed object replicas.

Multiprocessor real-time locking protocols for replicated resources.[4] In a real-time system, processors may not be the only resource tasks share. Many other resources in the system, such as memory objects and I/O devices, may also be shared among tasks. Algorithms designed for managing such non-processor shared resources are called *real-time locking protocols*.

Most real-time locking protocols support only non-replicated resources, *i.e.*, mutual exclusion is assumed for each lock. A few k-exclusion real-time locking protocols were designed for sharing replicated resources (Brandenburg and Anderson, 2011; Elliott and Anderson, 2011; Yang et al., 2013). However, all of them assume that each task may request only one replica at a time. In (Nemitz et al., 2016), we devised

[3]Types of data objects include simple data structures such as scalars, arrays, matrices, and images as well as higher-level data objects common to computer-vision algorithms such as histograms, image pyramids, and lookup tables.

[4]Details of this contribution have been published in the following paper:

Nemitz, C., Yang, K., Yang, M., Ekberg, P., and Anderson, J. (2016). Multiprocessor real-time locking protocols for replicated resources. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, pages 50–60.

multiprocessor real-time locking protocols for allocating replicated resources where individual tasks may request multiple replicas. We identified an allocation problem and an assignment problem related to the design of such a protocol and provided algorithms to solve each problem, respectively. Specifically, we presented a wait-free algorithm that can be applied to any replica-allocation protocol to solve the assignment problem. To solve the allocation problem, we presented both overhead-optimized and blocking-optimized replica-allocation protocols. For the overhead-optimized protocol, we provided a holistic blocking analysis technique that mitigates some of the pessimism in conventional blocking analysis; for the blocking-optimized protocol, we employed a cutting-ahead mechanism for which blocking is asymptotically optimal.

Uniprocessor mixed-criticality scheduling with permitted failure probability.[5] While the WCET abstraction plays an important role in the analysis of real-time systems, its estimation can be extremely difficult. As a result, even for a single piece of code, a broad range of estimates can be made for its WCET with different degrees of confidence. The higher the confidence, the greater the pessimism usually is as assumed in analysis, resulting in a larger WCET estimate. For some real-time systems, provisioning such WCETs by relatively pessimistic estimates may cause the underlying platform to be significantly under-utilized; on the other hand, provisioning such WCETs by relatively optimistic estimates may not cover the actual worst case and therefore violate safety requirements. As a solution, mixed-criticality (MC) scheduling has been proposed (Vestal, 2007).

In much work regarding MC scheduling (see (Burns and Davis, 2018) for a survey), tasks are categorized to two criticality levels, HI and LO. Each HI-task has two provisioned WCET estimates, a more pessimistic one and a more optimistic one. In contrast, each LO-task has one provisioned optimistic WCET only. The goal of MC scheduling is to allow both HI- and LO-tasks to complete in normal cases, while sacrificing LO-tasks for HI-tasks to complete in extreme cases. Thus, many MC scheduling algorithms drop all LO-jobs when any single HI-job overruns its optimistic WCET, so that every HI-job thereafter can be guaranteed to execute for up to its pessimistic WCET before its deadline. However, most such work was based on the assumption that *all* HI-tasks may have an active job overrunning its optimistic WCET *simultaneously*. This could be of extremely low probability that is not necessarily to be covered in the system design. In (Guo et al., 2015), we addressed this issue by introducing a new parameter, called *failure probability*, for each HI-task.

[5]Details of this contribution have been published in the following paper:

Guo, Z., Santinelli, L., and Yang, K. (2015). EDF schedulability analysis on mixed-criticality systems with permitted failure probability. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 187–196.

We then designed an EDF-based uniprocessor MC scheduling algorithm, under which the probability of missing any deadline of a HI-job is no greater than a specified *permitted system failure probability* for any system deemed schedulable by our presented schedulability tests.

8.3 Future Work

In this last section, we discuss several promising directions for future work following this dissertation.

Improving the tardiness bound under G-EDF. In Chapter 3, we proved a tardiness bound under preemptive G-EDF for scheduling a set of sporadic tasks on a uniform multiprocessor. However, neither this bound nor those derived by Devi (2006) and Erickson (2014) with respect to G-EDF scheduling on identical multiprocessors is known to be tight. That is, we were not able to construct a system in which observed maximum tardiness in a simulated schedule matches our analytical tardiness bound. In fact, we believe none of these bounds is actually tight. Future work could be directed at developing new frameworks and/or techniques to improve these bounds.

Tardiness bounds under global fixed-priority scheduling for npc-sporadic tasks. In most work regarding tardiness bounds, only dynamic-priority scheduling algorithms, such as G-EDF, are considered, perhaps because tardiness can easily grow without bound for global fixed-priority scheduling (Devi, 2006). However, this is true only if the conventional sporadic task model is assumed. In Chapter 5, we showed that, for the same task parameters, some task systems that are infeasible under the conventional sporadic task model can become feasible if the npc-task model is applied. Thus, we conjecture that tardiness is bounded under any fixed-priority scheduler for any feasible npc-sporadic task system. This conjecture needs future work to verify, and such work could be conducted for identical multiprocessors first and then extended to uniform multiprocessors.

Dynamic asymmetric multiprocessor platforms. As computing hardware continually evolves, it is not hard to foresee that processing platforms for many future real-time systems might be not only asymmetric but also dynamically varying during runtime. In fact, the dynamic voltage and frequency scaling (DVFS) technology may be able to allow speeds to vary during runtime. Furthermore, many embedded real-time systems are built with field programmable gate arrays (FPGAs), which are integrated circuits that can be configured to perform different functionalities. Some modern FPGAs are capable of *dynamic partial reconfiguration*, which allows one portion of an FPGA chip to be reconfigured while the rest keeps running.

This may result in a multiprocessor platform where processors can dynamically change their functionalities during runtime. As for virtualization, dynamically reallocating hardware resources for VPs is commonly supported by modern hypervisors. Future work could be directed at these scenarios.

Real-time locking protocols for asymmetric multiprocessors. Most existing real-time locking protocols were designed and analyzed for uniprocessors and symmetric multiprocessors. Such protocols and analyses are either inapplicable or overly pessimistic on asymmetric platforms. For example, considering that processors may have different speeds, the length of a critical section can be either scaled by the speeds (*e.g.*, regular CPU computation using a shared data object) or not scaled by the speeds (*e.g.*, accessing an external I/O device). While these two kinds of critical sections are considered the same in most existing real-time locking protocols and analyses, it intuitively may be better to treat them separately when different speeds are considered: scaled critical sections should tend to be scheduled on fast processors in order to reduce the time duration for which they may block other tasks; non-scaled critical sections should tend to be scheduled on slow processors in order to reserve fast processors to other tasks in need. Using this intuition to develop new real-time multiprocessor locking protocols and blocking analyses could be another direction for future work.

BIBLIOGRAPHY

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

Anderson, J., Bud, V., and Devi, U. (2005). 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.

Anderson, J., Erickson, J., Devi, U., and Casses, B. (2016). Optimal semi-partitioned scheduling in soft real-time systems. *Journal of Signal Processing Systems*, 84(1):3–23.

Andersson, B., Bletsas, K., and Baruah, S. (2008). Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 385–394.

Andersson, B. and Tovar, E. (2006). Multiprocessor scheduling with few preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334.

ARM (2018). Technologies—big.LITTLE. `http://www.arm.com/products/processors/technologies/biglittleprocessing.php`.

Bajaj, R. and Agrawal, D. (2004). Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118.

Baker, T. and Baruah, S. (2009). An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. *Real-Time Systems*, 43(1):3–24.

Baruah, S. (2014). Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 97–105.

Baruah, S. (2015a). The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition*, pages 1323–1328.

Baruah, S. (2015b). Federated scheduling of sporadic DAG task systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, pages 179–186.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., and Wiese, A. (2012). A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 63–72.

Baruah, S. and Carpenter, J. (2005). Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *Journal of Embedded Computing*, 1(2):169–178.

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

Baruah, S., Funk, S., and Goossens, J. (2003). Robustness results concerning edf scheduling upon uniform multiprocessors. *IEEE Transactions on Computers*, 52(9):1185–1195.

Bastoni, A., Brandenburg, B., and Anderson, J. (2011). Is semi-partitioned scheduling practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 125–135.

Bhatti, M., Belleudy, C., and Auguin, M. (2012). A semi-partitioned real-time scheduling approach for periodic task systems on multicore platforms. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1594–1601.

Bini, E., Bertogna, M., and Baruah, S. (2009a). Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 437–446.

Bini, E., Buttazzo, G., and Bertogna, M. (2009b). The multi supply function abstraction for multiprocessors. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 294–302.

Bletsas, K. and Andersson, B. (2009). Notional processors: An approach for multiprocessor scheduling. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12.

Bletsas, K. and Andersson, B. (2011). Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Systems*, 47(4):319–355.

Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., and Wiese, A. (2013). Feasibility analysis in the sporadic DAG task model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 225–233.

Brandenburg, B. and Anderson, J. (2011). Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and *k*-exclusion locks. In *Proceedings of the ACM International Conference on Embedded Software*, pages 69–78.

Brandenburg, B. and Gül, M. (2016). Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 99–110.

Burmyakov, A., Bini, E., and Tovar, E. (2014). Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376.

Burns, A. and Davis, R. (2018). Mixed criticality systems—a review. `https://www-users.cs.york.ac.uk/burns/review.pdf`.

Burns, A., Davis, R., Wang, P., and Zhang, F. (2012). Partitioned EDF scheduling for multiprocessors using a C=D scheme. *Real-Time Systems*, 48(1):3–33.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 886–893.

Dantzig, G. (1998). *Linear Programming and Extensions*. Princeton University Press, 11th edition.

Deng, Z. and Liu, J. (1997). Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 308–319.

Devi, U. (2006). *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Devi, U. and Anderson, J. (2008). Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189.

Dong, Z., Liu, C., Gatherer, A., McFearin, L., Yan, P., and Anderson, J. (2017). Optimal dataflow scheduling on a heterogeneous multiprocessor with reduced response time bounds. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems*, pages 15:1–15:22.

Durrieu, G., Faugère, M., Girbal, S., Pérez, D., Pagetti, C., and Puffitsch, W. (2014). Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software and Systems*.

Easwaran, A., Anand, M., and Lee, I. (2007). Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 129–138.

Easwaran, A., Shin, I., and Lee, I. (2009). Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59.

Elliott, G. and Anderson, J. (2011). An optimal *k*-exclusion real-time locking protocol motivated by multi-gpu systems. In *Proceedings of the 19th International Conference on Real-Time and Network Systems*, pages 15–24.

Elliott, G., Kim, N., Erickson, J., Liu, C., and Anderson, J. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Elliott, G., Yang, K., and Anderson, J. (2015). Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*, pages 273–284.

Erickson, J. (2014). *Managing Tardiness Bounds and Overload in Soft Real-Time Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Erickson, J. and Anderson, J. (2011). Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems*, pages 128–142.

Erickson, J., Anderson, J., and Ward, B. (2014). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems*, 50(1):5–47.

Erickson, J., Devi, U., and Baruah, S. (2010a). Improved tardiness bounds for global EDF. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 14–23.

Erickson, J., Guan, N., and Baruah, S. (2010b). Tardiness bounds for global EDF with deadlines different from periods. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems*, pages 286–301.

Fan, M. and Quan, G. (2012). Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 503–508.

Funk, S. (2004). *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC.

Funk, S. and Baruah, S. (2003). Characteristics of EDF schedulability on uniform multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 211–218.

Funk, S. and Baruah, S. (2005a). Restricting EDF migration on uniform heteroteneous multiprocessors. *Technique et Science Informatiques*, 24(8):917–938.

Funk, S. and Baruah, S. (2005b). Task assignment on uniform heterogeneous multiprocessors. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 219–226.

Funk, S., Goossens, J., and Baruah, S. (2001). On-line scheduling on uniform multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 183–192.

Grandpierre, T., Lavarenne, C., and Sorel, Y. (1999). Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 74–78.

Guan, N., Stigge, M., Yi, W., and Yu, G. (2010). Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 165–174.

Guo, Z., Santinelli, L., and Yang, K. (2015). EDF schedulability analysis on mixed-criticality systems with permitted failure probability. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 187–196.

Horvath, E., Lam, S., and Sethi, R. (1977). A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43.

Jiang, X., Guan, N., Long, X., and Yi, W. (2017). Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 80–91.

Jiang, X., Long, X., Guan, N., and Wan, H. (2016). On the decomposition-based global edf scheduling of parallel real-time tasks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 237–246.

Kato, S. and Yamasaki, N. (2007). Real-time scheduling with task splitting on multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450.

Kato, S. and Yamasaki, N. (2008). Portioned EDF-based scheduling on multiprocessors. In *Proceedings of the 8th ACM International Conference on Embedded Software*, pages 139–148.

Kato, S. and Yamasaki, N. (2009). Semi-partitioned fixed-priority scheduling on multiprocessors. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32.

Khronos Group (2014). The OpenVX™ specification. Version 1.0, Revision r28647, `https://www.khronos.org/registry/OpenVX/specs/1.0/OpenVX_Specification_1_0.pdf`.

Leoncini, M., Montangero, M., and Valente, P. (2017). A branch-and-bound algorithm to compute a tighter bound to tardiness for preemptive global EDF scheduler. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 128–137.

Leontyev, H. and Anderson, J. (2007a). Tardiness bounds for EDF scheduling on multi-speed multicore platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 103–111.

Leontyev, H. and Anderson, J. (2007b). Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–80.

Leontyev, H. and Anderson, J. (2009). A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43(1):60–92.

Leontyev, H. and Anderson, J. (2010). Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71.

Li, J., Agrawal, K., Lu, C., and Gill, C. (2013). Analysis of global EDF for parallel tasks. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 3–13.

Li, J., Ferry, D., Ahuja, S., Agrawal, K., Gill, C., and Lu, C. (2017). Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5):760–811.

Li, J., Saifullah, A., Agrawal, K., Gill, C., and Lu, C. (2014). Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 85–96.

Lipari, G. and Baruah, S. (2001). A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 26–35.

Lipari, G. and Bini, E. (2003). Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 151–158.

Lipari, G. and Bini, E. (2010). A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 249–258.

Liu, C. and Anderson, J. (2010). Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 3–13.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30(1):46–61.

Mercer, C., Savage, S., and Tokuda, H. (1994). Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 90–99.

Mok, A., Feng, X., and Chen, D. (2001). Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 75–84.

Nemitz, C., Yang, K., Yang, M., Ekberg, P., and Anderson, J. (2016). Multiprocessor real-time locking protocols for replicated resources. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, pages 50–60.

Parri, A., Biondi, A., and Marinoni, M. (2015). Response time analysis for G-EDF and G-DM scheduling of sporadic DAG-tasks with arbitrary deadline. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 205–214.

Patterson, J. and Chantem, T. (2016). EDF-hv: An energy-efficient semi-partitioned approach for hard real-time systems. In *Proceedings of the 24th International Conference on Real-Time and Network Systems*, pages 267–276.

Pinedo, M. (1995). *Scheduling, Theory, Algorithms, and Systems*. Prentice Hall.

Rainey, E., Villarreal, J., Dedeoglu, G., Pulli, K., Lepley, T., and Brill, F. (2014). Addressing system-level optimization with OpenVX graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 658–663.

Saifullah, A., Agrawal, K., Lu, C., and Gill, C. (2013). Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435.

Shin, I., Easwaran, A., and Lee, I. (2008). Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 181–190.

Shin, I. and Lee, I. (2003). Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 1–12.

Sousa, P., Souto, P., Tovar, E., and Bletsas, K. (2013). The carousel-EDF scheduling algorithm for multiprocessor systems. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 12–21.

Stafford, R. (2006). Random vectors with fixed sum. `http://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum`.

Stavrinides, G. and Karatza, H. (2011). Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques. *Simulation Modelling Practice and Theory*, 19(1):540–552.

Tong, G. and Liu, C. (2016). Supporting soft real-time sporadic task systems on heterogeneous multiprocessors with no uilitzation loss. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2740–2752.

Valente, P. (2016). Using a lag-balance property to tighten tardiness bounds for global EDF. *Real-Time Systems*, 52(4):486–561.

Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239–243.

Xu, M., Phan, L., Sokolsky, O., Xi, S., Lu, C., Gill, C., and Lee, I. (2015). Cache-aware compositional analysis of real-time multicore virtualization platforms. *Real-Time Systems*, 51(6):675–723.

Yang, K. and Anderson, J. (2014a). Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *Proceedings of the 12th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 30–39.

Yang, K. and Anderson, J. (2014b). Soft real-time semi-partitioned scheduling with restricted migrations on uniform heterogeneous multiprocessors. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 215–224.

Yang, K. and Anderson, J. (2015a). On the soft real-time optimality of global EDF on multiprocessors: From identical to uniform heterogeneous. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Yang, K. and Anderson, J. (2015b). An optimal semi-partitioned scheduler for uniform heterogeneous multiprocessors. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 199–210.

Yang, K. and Anderson, J. (2016a). On the dominance of minimum-parallelism multiprocessor supply. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 215–226.

Yang, K. and Anderson, J. (2016b). Tardiness bounds for global EDF scheduling on a uniform multiprocessor. In *Proceedings of the 7th International Real-Time Scheduling Open Problems Seminar*, pages 3–4.

Yang, K. and Anderson, J. (2017). On the soft real-time optimality of global EDF on uniform multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 319–330.

Yang, K., Elliott, G., and Anderson, J. (2015). Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, pages 77–86.

Yang, K., Yang, M., and Anderson, J. (2016). Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 349–358.

Yang, M., Lei, H., Liao, Y., and Rabee, F. (2013). PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi-gpu sharing under partitioned scheduling. In *Proceedings of the 11th IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 207–214.